ANOMALY DETECTION IN IAC SCRIPTS TO PREVENT CLOUD MISCONFIGURATIONS USING UNSUPERVISED LEARNING

Noor Elahi Ali Shibly
172-35-2204

Supervised by
Dr. Imran Mahmud
Professor & Head
Department of Software Engineering
Daffodil International University

Department of Software Engineering

DAFFODIL INTERNATIONAL UNIVERSITY

©Daffodil International University

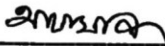
APPROVAL

This thesis titled on "**Anomaly Detection in IaC Scripts to Prevent Cloud Misconfigurations Using Unsupervised Learning**", submitted by **Noor Elahi Ali Shibly (ID: 172-35-2204)** to the Department of Software Engineering, Daffodil International University has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering and approval as to its style and contents.

BOARD OF EXAMINERS

Chairman

Dr. Imran Mahmud
Professor & Head
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University



Internal Examiner 1

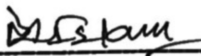
Afsana Begum
Assistant Professor
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University

Internal Examiner 2

Md. Shohel Arman
Assistant Professor
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University

Internal Examiner 3

Nadira Islam
Assistant Professor
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University



External Examiner

Md Manowarul Islam
Professor
Department of Computer Science and Engineering
Jagannath University, Bangladesh

©Daffodil International University

THESIS DECLARATION LETTER (OPTIONAL)

Librarian,
Daffodil International University,
Daffodil Smart City,
Ashulia. Dhaka, Bangladesh

Dear Sir,

CLASSIFICATION OF THESIS AS RESTRICTED

Please be informed that the following thesis is classified as RESTRICTED for a period of three (3) years from the date of this letter. The reasons for this classification are as listed below.

Author's Name: Noor Elahi Ali Shibly

Thesis Title: Anomaly Detection in IaC Scripts to Prevent Cloud Misconfigurations Using Unsupervised Learning

Reasons:

(i)

(ii)

(iii)

Thank you.

Yours faithfully,

____

(Supervisor's Signature)

Date: 26 December 2025

Stamp:

SUPERVISOR'S DECLARATION

I hereby declare that I have checked this thesis and, in my opinion, this thesis is adequate in terms of scope and quality for the award of the degree of Bachelor of Science.

____

(Supervisor's Signature)

Full Name : Professor Dr. Imran Mahmud

Position : Head of the Department, Department of Software Engineering

Date : 26 December 2025

STUDENT'S DECLARATION

I hereby declare that the work in this thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at Daffodil International University or any other institution.

Shibly

(Student's Signature)

Full Name : Noor Elahi Ali Shibly

ID Number : 172-35-2204

Date : December 26, 2025

Anomaly Detection in IaC Scripts to Prevent Cloud Misconfigurations Using Unsupervised Learning

Noor Elahi Ali Shibly
172-35-2204

Thesis submitted in fulfillment of the requirements
for the award of the degree of
Bachelor of Science/Master of Science

Department of Software Engineering (Major in Cyber Security)

DAFFODIL INTERNATIONAL UNIVERSITY

December 2025

©Daffodil International University

ACKNOWLEDGEMENTS

First of all, I am thankful to Almighty Allah for providing me with the ability and desire to finish this thesis. I want to thank my supervisor and inspiration, Professor Dr. Imran Mahmud, for his support, direction, and inspiration throughout the whole study. The support and insightful comments played a vital role in shaping this research. I am also thankful to all of my teachers who have contributed to my academic development and guided me to where I am today.

- Noor Elahi Ali Shibly

DEDICATION

It is hereby declared that this thesis, including all the research-based experimental work, has been completed by me under the supervision of Professor Dr. Imran Mahmud, Head of the Department of Software Engineering, Daffodil International University. I also declare that neither this thesis nor any part of the research-based work has been submitted elsewhere for the award of any degree

(Student's Signature)

Full Name : Noor Elahi Ali Shibly

ID Number : 172-35-2204

Date : December 26, 2025

ABSTRACT

Poorly configured Infrastructure as Code (IaC) is still a major problem in cloud security, particularly because conventional rule-based tools may frequently fail to see hidden, sophisticated, or even entirely novel risks. We present four unsupervised machine-learning algorithms in this work, which are meant to identify suspicious or unsafe patterns in Terraform configuration files, specifically AWS S3 security. The system identifies and processes each Terraform file by extracting 22 features related to security, and these features are used by four different anomaly-detection models, including Isolation Forest, One-Class SVM, Autoencoder, and Local Outlier Factor. Their results are then pooled together to form an ensemble and ensure the conclusion becomes more accurate. Our findings reveal that, besides performing well, particularly with the autoencoder and the ensemble, this method is also more efficient in detecting problems than the traditional tools of the trade in the field of static analysis. The framework gives clear and explainable feedback to enable the developers to know the reason why something was flagged. On the whole, the research demonstrates that the concept of unsupervised learning can provide a viable and scalable means of identifying IaC misconfigurations prior to their manifestation in actual security issues in the real world.

Keywords: Infrastructure as Code, Cloud Misconfigurations, Terraform, Unsupervised Machine Learning, Anomaly Detection, AWS S3, Isolation Forest, Autoencoder, DevSecOps, CI/CD Security

TABLE OF CONTENT

DECLARATION

TITLE PAGE

ACKNOWLEDGEMENTS ii

Dedication iii

ABSTRACT iv

TABLE OF CONTENT v

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF SYMBOLS xiii

LIST OF ABBREVIATIONS xvii

LIST OF APPENDICES xx

CHAPTER 1 INTRODUCTION 1

1.1 Infrastructure as Code: Strengths and Weaknesses 1

1.2 Real-World Consequences of IaC Misconfigurations 2

1.3 Current Tools for Detecting IaC Security Issues 3

1.4 Machine Learning in Configuration Validation 4

1.5 Gaps in Existing Research 4

1.6 Objectives of This Study 5

1.7 Summary 6

CHAPTER 2 LITERATURE REVIEW 8

2.1 Common Cloud Misconfigurations 8

2.1.1 Access Control and Identity Management Misconfigurations 9

2.1.2 Access Key Misuse and Hard-Coded Secrets 9

2.1.3 Cognito Misconfigurations 10

2.1.4 IP Address Binding 10
2.1.5 S3 Bucket Misconfigurations 11
2.1.6 EBS Encryption Misconfiguration / Encryption at Rest 12
2.1.7 Encryption in Transit 12
2.1.8 Security Group Misconfiguration 13
2.1.9 General Configuration Data Issues 13
2.1.10 Logging and Monitoring Issues 13
2.2 Real-World Consequences of Cloud Misconfigurations 14
2.2.1 Data Breaches and Data Leakage 14
2.2.2 Malicious Code Injection 15
2.2.3 Service Disruption and Outage 15
2.2.4 Unauthorized Access and Lateral Movement 16
2.2.5 Compromise of Entire Cloud Infrastructure 16
2.2.6 Hindered Reliability of Automated Deployment Pipelines 17
2.2.7 Non-Compliance with Regulatory Standards 17
2.3 Cloud Providers and Services Most Vulnerable to Misconfigurations 18
2.3.1 Comparative Misconfiguration Trends Across Providers 18
2.3.2 Cloud Services Most Prone to Misconfiguration 19
2.3.3 Real-World Case Studies of Misconfigurations 20
2.3.4 Provider-Specific Mitigation Features 21
2.4 Security Smells in Infrastructure as Code (IaC) 22
2.4.1 Admin by Default 23
2.4.2 Empty Password 23
2.4.3 Hard-Coded Secret 23
2.4.4 Invalid IP Address Binding 24
2.4.5 Suspicious Comment 24

2.4.6 Use of HTTP Without TLS 25

2.4.7 Use of Weak Cryptographic Algorithms 25

2.5 Effectiveness of Tools for Detecting Security Smells in Infrastructure as Code 26

2.5.1 SLIC: Security Linter for Infrastructure as Code 26

2.5.2 SecureCode 28

2.5.3 SLAC: Security Linter for Ansible and Chef 28

2.6 Impact of Security Smells on IaC Security 29

2.6.1 Security Risk and Vulnerability Introduction 29

2.6.2 Real-World Consequences and Failures 30

2.6.3 Persistence of Smells in Codebases 30

2.6.4 Frequency and Severity in Practice 30

2.6.5 Hindrance to Reliability and Maintainability 31

2.7 More Tools for Detecting Security Smells and Misconfigurations in Infrastructure as Code 31

2.7.1 Checkov 32

2.7.2 SecureCode 33

2.7.3 SLIC (Security Linter for Infrastructure as Code Scripts) 34

2.7.4 Additional Tools for IaC Static Analysis 35

2.8 Unsupervised Machine Learning for Anomaly Detection in Cloud Security and IaC 37

2.8.1 Applied Unsupervised Learning Methods 37

2.8.2 Feature Extraction Techniques 38

2.8.3 Performance of Unsupervised Methods 39

2.8.4 Challenges, Limitations, and Advantages 40

2.8.5 Specific Implementations and Evaluations 41

2.9 Summary 41

CHAPTER 3 METHODOLOGY **43**

3.1 Overview of the Proposed System 43

3.2 Data Collection and Dataset Preparation 45

3.2.1 Terraform Configuration Dataset 45

3.2.2 Terraform Parsing Process 46

3.2.3 Data Pre-processing and Normalization 47

3.3 Feature Extraction and Engineering 48

3.3.1 Feature Definition and Extraction Strategy 48

3.3.2 Mathematical Feature Representation 49

3.3.3 Derived Security Metrics 49

3.4 Unsupervised Anomaly Detection Models 50

3.4.1 Isolation Forest 51

3.4.2 One-Class Support Vector Machine 52

3.4.3 Autoencoder Neural Network 53

3.4.4 Ensemble Aggregation Strategy 54

3.5 Training and Validation Strategy 55

3.6 Anomaly Scoring and Threshold Determination 55

3.7 Visualization and Explainability Pipeline 56

3.7.1 Dimensionality Reduction for Visualization 57

3.7.2 Visualization Outputs 58

3.8 Performance Evaluation Metrics 58

3.9 System Implementation and Integration 60

3.10 Summary 60

CHAPTER 4 RESULTS AND DISCUSSION **62**

4.1 Experimental Setup and Dataset Characteristics 62

	4.1.1 Dataset Composition	63
	4.1.2 Model Configuration	64
4.2	Model Performance Evaluation	65
	4.2.1 ROC Curve Analysis	65
	4.2.2 Precision-Recall Analysis	68
	4.2.3 Error Analysis	70
	4.2.4 Anomaly Score Distribution Analysis	71
4.3	Model-Specific Performance Analysis	72
	4.3.1 Ranking and Comparing Models Using Overall Performance	72
	4.3.2 Isolation Forest Performance	73
	4.3.3 Autoencoder Performance	74
	4.3.4 One-Class SVM Performance	75
	4.3.5 Local Outlier Factor Performance	76
	4.3.6 Ensemble Model Performance	78
4.4	Statistical Significance Testing	80
4.5	Feature Space Visualization and Interpretability	80
	4.5.1 Feature Importance Analysis	80
4.6	Real Misconfiguration Detection Examples	82
4.7	Comparison with Existing Approaches	83
4.8	Discussion and Implications	85
	4.8.1 Validation of Research Objectives	85
	4.8.2 Implications for Cloud Security	86
	4.8.3 Limitations and Threats to Validity	86
	4.8.4 Lessons Learned	87
4.9	Summary	88

CHAPTER 5 CONCLUSION **90**

APPENDICES **91**

APPENDIX A: FEATURE DATA AND RAW DATA. **91**

APPENDIX B: DIMENSIONALITY REDUCTION VISUALIZATIONS **93**

REFERENCES **96**

LIST OF TABLES

Table 1: Extracted Feature Taxonomy for IaC Analysis 48

Table 2: Dataset Summary and Composition 62

Table 3: Model Hyperparameters for Anomaly Detection 64

Table 4: Comprehensive Performance Metrics for Anomaly Detection Models 67

Table 5: Confusion Matrix Summary for Test Set (n=10) 70

Table 6: All Models Ranked Based Upon Overall Performance 72

Table 7: Statistical Significance Testing via McNemar’s Test 80

Table 8: Top 10 Feature Importance Rankings 81

Table 9: Real Misconfiguration Detection Examples 83

Table 10: Comparison with Existing IaC Security Tools 83

LIST OF FIGURES

Figure 1:	The overall architecture of the system with the five primary blocks.	44
Figure 2:	The Terraform parsing process.	46
Figure 3:	Machine learning model workflow.	51
Figure 4:	Visualization pipeline.	57
Figure 5:	ROC curve comparison across all anomaly detection models.	66
Figure 6:	Precision-Recall Curve Comparison illustrating each model's ability to detect misconfigurations within the imbalanced data set.	69
Figure 7:	Histogram of the normalized anomaly score distributions for each model	71
Figure 8:	Anomaly Detection Visualization Using Isolation Forest in 2D PCA Space.	73
Figure 9:	Autoencoder Anomaly Detection Results in PCA-Projected Space.	74
Figure 10:	Results of a One-Class SVM show moderate performance in detecting IaC misconfiguration.	75
Figure 11:	Predictions from the Local Outlier Factor algorithm are so riddled with error that it cannot provide a reliable classification	77
Figure 12:	Ensemble model predictions from all four models using weighted score combination.	79

LIST OF SYMBOLS

$\mathbb{R}$	The set of Real Numbers
$\|\cdot\|$	The norm of a vector or matrix
argmin	The argument of the minimum
argmax	The argument of the maximum
$\mathcal{D}$	The complete dataset of Terraform scripts, represented as $\{d_1, d_2, \dots, d_N\}$
d_i	A single Terraform script (sample) in the dataset
$\mathcal{D}_{\text{correct}}$	Subset of the dataset containing correctly configured scripts
$\mathcal{D}_{\text{misconfig}}$	Subset of the dataset containing misconfigured scripts
N	The total number of samples in the dataset
n	The number of training samples
$P(d_i)$	The parsing function that transforms d_i into a structured representation
r_j	A parsed resource block within a Terraform script
A_j	The set of attributes associated with a parsed resource block r_j
M	The dimensionality of the feature space (number of features)
X	The complete feature matrix ($X \in \mathbb{R}^{N \times M}$)
X_{raw}	The raw feature matrix extracted before normalization
$\mathbf{x}$	A generic feature vector representing a single sample in the model
$\hat{\mathbf{x}}$	The reconstructed version of the input $\mathbf{x}$ from the Autoencoder
Φ	The feature extraction function that maps a script to its feature vector

$\Phi(d_i)$ or $\mathbf{f}_i$	The feature vector for d_i, represented as $[f_{i1}, f_{i2}, \dots, f_{iM}]^T$
f_{ij} or x_{ij}	The value of the j-th feature for the i-th sample
x_{ij}^{median}	The median value of feature j (used for imputation)
Q_1, Q_2, Q_3	The first quartile, median (second quartile), and third quartile, respectively
S	The set of security features used in the Security Score calculation
w_j	Uniform weights applied to features f_{ij} in the Security Score
ϵ	A variable from a uniform distribution $\mathcal{U}(0,2)$, introducing controlled stochasticity
$\mathcal{U}(a,b)$	Uniform distribution between a and b
ν	The contamination parameter (expected proportion of anomalies)
$s(\mathbf{x}, n)$	The anomaly score for a sample $\mathbf{x}$
$s_k(\mathbf{x})$	The anomaly score from model k
$s_{\text{ensemble}}(\mathbf{x})$	The final ensemble anomaly score
w_k	The weight assigned to model k in the ensemble
$h(\mathbf{x})$	The path length of sample $\mathbf{x}$ in an Isolation Tree
$E[h(\mathbf{x})]$ or $E\big(h(\mathbf{x})\big)$	The average path length of $\mathbf{x}$ across all Isolation Trees
$c(n)$	A normalization factor for path length
$H(i)$	The harmonic number, calculated as $\ln(i) + \gamma$
γ	Euler's constant (used in the harmonic number calculation)
$\mathbf{w}$	The weight vector defining the separating hyperplane in One-Class SVM
ρ	The offset parameter for the hyperplane in One-Class SVM
$\boldsymbol{\xi}$	The slack variables in the One-Class SVM optimization problem

$\phi(\cdot)$	The kernel mapping function in SVM
$K(\cdot,\cdot)$	The kernel function (e.g., RBF kernel)
α_i	The Lagrange multipliers in the SVM dual problem
$E(\cdot)$	The encoder network in an Autoencoder
$D(\cdot)$	The decoder network in an Autoencoder
D	The dimensionality of the Autoencoder's bottleneck layer
$\mathcal{L}(\mathbf{x})$	The reconstruction loss for a sample $\mathbf{x}$ in the Autoencoder
θ	The classification threshold for the anomaly score
$Q_{1-\nu}$	The $(1-\nu)$-th quantile of the anomaly score distribution (which defines θ)
$\hat{y}(\mathbf{x})$	The predicted class label for sample $\mathbf{x}$ (1 for anomaly, 0 otherwise)
$\mathbf{z}_i$	The 2D projection of sample $\mathbf{x}_i$ after Principal Component Analysis (PCA)
W	The projection matrix (top two eigenvectors) for PCA
$\boldsymbol{\mu}$	The mean vector of the feature space
Σ	The covariance matrix of the feature space
$\mathrm{KL}(P \parallel Q)$	The Kullback–Leibler divergence between distributions P and Q (used in t-SNE)
TP	True Positives
FP	False Positives
FN	False Negatives
TN	True Negatives
TPR	True Positive Rate (Recall)
FPR	False Positive Rate
AUC	Area Under the ROC Curve
AP	Average Precision (Area Under the Precision-Recall Curve)

$P(k)$	Precision at threshold k
$R(k)$	Recall at threshold k

LIST OF ABBREVIATIONS

ACL	Access Control List
ADTK	Anomaly Detection Tool Kit
AE	Autoencoder (used in figures/tables)
API	Application Programming Interface
AST	Abstract Syntax Tree
AUC	Area Under the Curve
AWS	Amazon Web Services
CI/CD	Continuous Integration/Continuous Deployment
CIS	Center for Internet Security (referenced in benchmarks)
CLI	Command-Line Interface
CNN	Convolutional Neural Network
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DevSecOps	Development, Security, and Operations
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
FN	False Negative
FP	False Positive
FPR	False Positive Rate
GCP	Google Cloud Platform
GDPR	General Data Protection Regulation
HCL	HashiCorp Configuration Language
HIPAA	Health Insurance Portability and Accountability Act
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IAM	Identity and Access Management
IBM	International Business Machines (company reference)
ID	Identifier
IF	Isolation Forest (used in figures/tables)

IP	Internet Protocol
JSON	JavaScript Object Notation
LLM	Large Language Model
LOF	Local Outlier Factor
LSTM	Long Short-Term Memory
MD5	Message-Digest Algorithm 5
ML	Machine Learning
MLP	Multi-Layer Perceptron
NaN	Not a Number
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PCA	Principal Component Analysis
PII	Personally Identifiable Information
PIPEDA	Personal Information Protection and Electronic Documents Act
PR	Precision-Recall
PR-AUC	Precision-Recall Area Under the Curve
RBF	Radial Basis Function
REPL	Read-Eval-Print Loop
ROC	Receiver Operating Characteristic
ROC-AUC	Receiver Operating Characteristic Area Under the Curve
RNN	Recurrent Neural Network
S3	Simple Storage Service
SDK	Software Development Kit
SHA-1	Secure Hash Algorithm 1
SLAC	Security Linter for Ansible and Chef
SLIC	Security Linter for Infrastructure as Code
SOM	Self-Organizing Map
SVM	Support Vector Machine
TF	Terraform (also used in figures/tables)
TLS	Transport Layer Security
TN	True Negative
TP	True Positive
TPR	True Positive Rate

t-SNE	t-Distributed Stochastic Neighbor Embedding
VPC	Virtual Private Cloud
WAF	Web Application Firewall

LIST OF APPENDICES

Appendix A: Feature Data And Raw Data 88

A.1 Feature Names 88

A.2 Sample of Raw Features 89

A.3 Feature Distribution Visualization 90

Appendix B: Dimensionality Reduction Visualizations 91

B.1 Principal Component Analysis (PCA) Visualization 91

B.2 t-Distributed Stochastic Neighbor Embedding (t -SNE) Visualization. 92

CHAPTER 1

INTRODUCTION

Cloud computing has radically transformed the current computing paradigms, which have made it easy to have elastic and scalable infrastructure that sustains a variety of enterprise applications. Although changing to cloud-based environments has many benefits, there are some complicated security issues. Among them, cloud misconfigurations are one of the most common causes of security breaches, which can be caused by errors in the settings of cloud services or insufficient adherence to the best practices [1]. These malpractices are augmented by the emergence of Infrastructure as Code (IaC), a development that enables infrastructure provisioning via scripting written in Infrastructure as Code languages, including Terraform, Ansible, and AWS CloudFormation.

IaC increases cloud infrastructure efficiency, reproducibility, and automatization. It also increases the risk, however, because it allows to easily reproduce risky settings on a canvas wide level. The mistakes that are created during one IaC script can be extended to numerous deployments and this can put organizations at serious risk [3].

The impact of cloud misconfigurations has been noted in several studies as well as in the real-world breaches. Examples of incidents can be found in the Capital One breach (2019), the Twilio SDK breach (2020), or the Imperva data leak (2019) that demonstrate how a combination of such simple misconfigurations (such as open S3 buckets, incorrect IAM roles, or exposed credentials) can cause tremendous amounts of data to be lost or breached by criminals [1].

1.1 Infrastructure as Code: Strengths and Weaknesses

IaC yields agile and scalable infrastructure management that is version-controlled. It allows suitable deployments and the development and operation teams to collaborate. The codification of infrastructure, however, also assigns security risks to the developers, who are not necessarily domain experts in security.

Security smells are empirically determined patterns in the IaC code to indicate insecure design. As an illustration, Rahman et al. [3] have recognized seven security smells, namely: "admin by default," hard-coded secret, empty password, invalid IP address binding, use of HTTP without TLS, suspicious comments, and use of weak cryptographic algorithms. Thousands of open-source repositories have been found to have these smells and some have lived as long as 98 months [3].

Even though the smells are not vulnerabilities in themselves, they increase the probability of misconfiguration considerably. The tools like SLIC (Security Linter for Infrastructure as Code) have been created in order to detect such patterns with a high level of accuracy and recall when used on Puppet scripts [3].

1.2 Real-World Consequences of IaC Misconfigurations

The effect of cloud misconfigurations is dire. The capital one breach took advantage of an instance of EC2 that had high IAM permissions and a misplaced firewall that allowed attackers to have access to sensitive S3 data [1]. In a parallel example, in the Twilio SDK attack, the insecurely set up cloud storage allowed attackers to inject malicious code into distributed software packages, impacting thousands of down-stream systems [1].

The other examples are the Imperva breach, with unsecured databases, and the Facebook outage of October 2021, when a single configuration mistake impacted the services of 2.9 billion users around the world [1]. These instances indicate that not only misconfigurations are a threat to security but also affect availability, reliability, and compliance.

Reliability of the CI/CD pipelines is also impeded by misconfigurations. As an illustration, malfunctioning code was the cause of hundreds of user directories being removed in Wikimedia Commons, as Rahman and Williams have explained [3].

The size of the issue is overwhelming. According to Guffey and Li, 99% of enterprise cloud erroneous setup is not reported, and 45% of the data breaches between March 2021 and March 2022 were related to cloud surroundings [1].

1.3 Current Tools for Detecting IaC Security Issues

Several tools have been developed to identify misconfigurations and security smells in IaC scripts:

- **Checkov**: An open-source tool which scans Terraform and CloudFormation scripts for compliance with security policies like the CIS benchmarks [1].

- **TFLint**: Focuses on Terraform-specific linting and potential security violations[1].

- **SLIC**: Detects predefined security smells in Puppet scripts and has been validated across multiple repositories [3].

- **SecureCode**: Scan embedded Shell/PowerShell code in Ansible playbooks and detect operational risks in terms of severity categories, such as security, availability and reliability [4].

- **SLAC**: Extension of SLIC to support Ansible and Chef including rule-based and machine learning based detection mechanisms [5]

Despite their contributions, these tools rely heavily on static rules and pattern matching, which limits their ability to generalize across different IaC languages and identify previously unknown misconfigurations. Additionally, they tend to generate false

positives or fail to detect subtle changes in misconfigurations when variables and context specific logic are involved [5].

1.4 Machine Learning in Configuration Validation

Recent attempts have been made using machine learning, specifically large language models (LLMs), for the validation of configurations. Lian et al. proposed Ciri, an LLM-based configuration validation system that takes advantage of few-shot learning to identify misconfigurations using little labeled data [2]. While promising, this approach suffers from the tendency of LLMs to hallucinate results and their bias to common configurations.

In contrast, unsupervised learning methods (Isolation Forest, Autoencoders, One-Class SVM, etc.) do not require labelled data and are capable of detecting anomalies according to the statistical difference from the expected behaviours. These methods have shown promise in the detection of system anomalies in the cloud environments [10].

Applying such methods to IaC scripts includes feature extraction - translating the structure, syntax and metadata of code into vectors which can be fed into ML models. Techniques like abstract syntax trees, n-gram tokenization and code embeddings are key to facilitate meaningful learning [7].

1.5 Gaps in Existing Research

Even though some academic literature applied machine learning in the context of runtime monitoring, only a small number of studies applied it to the analysis of the static IaC scripts. The majority of the detection mechanisms remain rule-based and ML applications are yet to be incorporated fully in DevOps working processes.

Another area that is being overlooked is explainability. Developers are more likely to trust and take action on a detection result if it contains clear, actionable

recommendations. Models that only detect "anomalies" without an explanation have limited usability in real world pipelines.

In addition, the number of frameworks that can assist in cross-language IaC analysis, i.e. be equally useful with Terraform, Ansible and CloudFormation, is low. Most tools like SLIC are keywords to certain syntax constructions, and such tools reduce their scalability [3][5].

1.6 Objectives of This Study

The objective of this research is to address these challenges and create an automated system to identify misconfigurations and anomalies in IaC scripts with the use of an unsupervised learning approach to machine learning. The system will be fully pipelineed - the features will be extracted, the anomalies will be determined, and their explanation will be achieved - and it will be tested regarding its accuracy and usability.

The specific objectives of this study are:

1. To identify common misconfigurations in IaC (Infrastructure-as-Code) scripts that lead security flaws in Cloud Service.
2. To review the tools and methods (e.g., Checkov, TFLint, and others) that are being used to detect IaC vulnerabilities.
3. To design an automated system that scans and analyzes IaC scripts for unusual or risky configurations.
4. To detect anomalies apply unsupervised machine learning techniques (e.g., Isolation Forest, and others).
5. To mitigate risks of cloud breaches by alerting developers or DevOps engineers before deployment.
6. To create a pipeline for efficient feature extraction from IaC scripts, which can provide suitable representation for anomaly detection.

7. To compare different unsupervised machine learning techniques (e.g., Isolation Forest, Autoencoders) to evaluate the effectiveness in the terms of detection accuracy and performance.

8. To increase the explainability and usability of the system by explaining and recommending detected anomalies to developers.

1.7 Summary

The most frequent cause of cloud security breaches are misconfigured clouds; usually caused by errors from humans or poor security practices in Infrastructure as Code (IaC). IaC enables you to automate cloud infrastructure, provide scalability and consistency. However, with IaC if there is a misconfiguration made in IaC script, that configuration will then replicate across all of environments. Misconfigurations have resulted in significant breaches including Capital One and Imperva breaches. These breaches occurred due to simple errors such as open storage buckets and/or insecure IAM roles. While tools such as Checkov, TFLint and SLIC assist in identifying security issues they have limited accuracy and cannot support multiple languages and can't detect previously unknown misconfigurations.

This research therefore proposes developing an automated anomaly detection system utilizing unsupervised machine learning techniques to automatically find anomalous IaC scripts. Anomaly detection algorithms such as Isolation Forests and Autoencoders can be used to identify anomalies in data without requiring any labeled training data; thus providing a more generalized and flexible solution than traditional supervised machine learning techniques. In addition to detecting anomalies, the proposed system consists of three primary components: feature extraction, anomaly detection and explanation. Feature extraction identifies which features of a configuration are contributing to the risk associated with the configuration, while the anomaly detection component identifies whether those features represent a misconfiguration. Finally, the explanation component explains why a particular configuration was identified as being risky, enabling developers to understand what changes need to be made to mitigate the risk. Ultimately, this approach should improve the reliability of the cloud based on the

fact that misconfigurations will be detected earlier in the development process and strengthen DevSecOps processes.

CHAPTER 2

LITERATURE REVIEW

This literature review intends to discuss the existing studies and research towards cloud security, Infrastructure as Code (IaC) and Machine Learning for Anomaly Detection. Cloud computing has become an essential technology for organizations, however, it also presents new security challenges, specifically configuration errors in cloud services. IaC, a way of automating the configuration of the cloud with code, has become a popular method but it is also subject to security issues, such as misconfigurations and security smells.

To overcome these challenges, researchers have looked into various tools and techniques for detecting and mitigating IaC security risks. This review will give an overview of cloud misconfigurations, describe the concept of IaC and the security issues, discuss the existing tools for IaC security analysis and discuss the application of machine learning techniques to detect anomalies in IaC scripts. The review will also highlight the gaps in the current research, as the basis for the development of an automated anomaly detection system for IaC scripts using unsupervised learning.

2.1 Common Cloud Misconfigurations

Misconfigurations in the cloud environment is one of the most important and ongoing security issues, especially when using Infrastructure as Code (IaC) to manage cloud infrastructure. Misconfigured resources can result in exposing sensitive data, enabling privilege escalation, or even total system compromise. The following discussion is a synthesis of the most prevalent types of misconfigurations that have been reported across cloud platforms as documented in current research.

2.1.1 Access Control and Identity Management Misconfigurations

One of the most common security problems found is the misconfiguration of Identity and Access Management or IAM policies. According to Guffey and Li [1], IAM role grants and policies such as the use of wild cards or administrator roles that are overly permissive violate the principle of least privilege and pose a great risk of compromise. If an attacker gains access to an identity with too many permissions, it can result in privilege escalation and lateral movement in the cloud environment. The complexity of cloud native architectures only serves to make enforcing least privilege even more difficult.

The "admin by default" pattern, which is when users or services are automatically granted high-privilege roles without any explicit need, is known to be a common security smell in IaC codebases (Borovits et al. [7]). A good example of this category of misconfiguration is the Capital One breach in 2019 in which an attacker exploited IAM policies attached to an EC2 instance to access confidential S3 data [Guffey and Li [1]. A number of tools have been written to spot such IAM weaknesses in cloud environments [1] (CloudStrike).

Verdet et al. [14] found access policy misconfigurations are one of the most frequently fixed security categories within IaC templates across AWS, Azure and GCP. Despite this, improper implementation is done in almost one-third of such policies. Furthermore, it seems that the amount of effort needed to prevent "admin by default" privileges also varies from provider to provider, with Azure cited as being less conducive to secure IAM defaults compared to AWS and GCP [14].

2.1.2 Access Key Misuse and Hard-Coded Secrets

The improper handling of access keys, especially hard coding them into IaC scripts or applications is a significant threat. As Guffey and Li [1] explain, access credentials such as AWS API keys are often embedded in source code or sitting on exposed compute instances, which results in unauthorized access if discovered. This problem is very common in IaC configurations in all the major cloud platforms (Verdet et al. [14]).

While not all cases of security smells are listed with specific numbers in the mentioned works, the existence of hard-coded secrets is repeatedly stated to be widespread. Guffey and Li [1] point to the Imperva breach in 2019 for an example, where an exposed API-key on an unprotected EC2 instance caused access to sensitive systems. Tools such as S3crets Scanner are meant to find such secrets in misconfigured storage systems. Verdet et al. [14] report that AWS and Azure have a superior implementation of policies that prevent hard coded secrets compared to GCP.

2.1.3 Cognito Misconfigurations

Another cause of possible misconfiguration is AWS Cognito, a service used to manage users for identity and access management. As described by Guffey and Li [1], access tokens issued by Cognito may be misused to interact directly with AWS services using the command-line interface (CLI). This ability, if not guarded, might open up the backbone's user pool properties or a backend service to an unauthorised user, leading to injection attacks. Although in the paper, Cognito is listed as one of the 6 commonly studied misconfiguration types, no publicly disclosed breach has been directly attributed to this problem.

2.1.4 IP Address Binding

Misconfigurations that are easy to detect and have a significant impact on control of network access include improper set up of IP address bindings (where it is not restricted). According to Guffey and Li [1], most security groups are open to access by any IP address (e.g., 0.0.0.0/0) and this significantly increases the attack space. Verdet et al. [14] categorize this as a type of policy in IaC, dubbed as IP address binding that are supposed to implement network-level least privilege. Nonetheless, despite the relative ease with which these restrictions can be added in code, approximately one-third of the configurations that they tested could not apply them.

The Capital One leak resulted from an initial breach through an improperly set firewall or security group, and this breach highlights the possible effects of lax IP restrictions. Guffey and Li [1] stated that 437 out of 440 default VPC security group

configurations failed basic access control checks. These failures are directly connected to IP address binding which is improperly enforced (Verdet et al. [14]).

2.1.5 S3 Bucket Misconfigurations

Most S3 bucket misconfigurations remain a major security risk of bulk clouds, and many high-profile attacks are based on those vulnerabilities. According to Madnick [19], the leading cause of data breaches is cloud misconfigurations, including those involving publicly accessible S3 buckets where everyone can see or edit the information stored in them by merely having a URL. This gives opportunity to attackers to utilize the sensitive information which results in data exfiltration, leakage or even malicious payload injection.

In the case of the Capital One breach of 2019, the breach was directly linked to a misconfigured WAF (web application firewall) and IAM permissions which gave an intruder access to S3 buckets containing sensitive customer information. Correspondingly, the breach of Twilio in 2020 was further aggravated by poor settings of cloud storage, which allowed hackers to steal sensitive data.

In addition to these occurrences, Dikshanta and Verma [20] state that the misconfigurations of S3 buckets have been an issue since the emergence of cloud security. Since they have been working on the way of increasing false positive reduction on S3 public access detection it is clear that despite such tools as S3Scanner are needed to detect the misconfiguration the problem is still widespread because most default configurations are more complex and nobody is supervising the transition of clouds in the proper way.

Even though AWS has made recent advances in default S3 policies, which have made S3 buckets less susceptible to misconfigurations, the issue still exists. According to Verdet et al. [14], Infrastructure as Code (IaC) deployments commonly have misconfigured S3 buckets because of the auto-provisioning of clouds, which commonly forces extremely wide access controls into production without sufficient validation.

In addition, as noted by IBM and Ponemon [21], breaches related to misconfigurations are costly, especially in sectors such as healthcare and finance, where S3 bucket misconfigurations are commonly used to breach large amount of personal identifiable information (PII). Their report of 2025 emphasizes the significance of security tools that are supported by AI to identify these vulnerabilities in their initial stages, thus minimizing breach expenses and time of containment.

2.1.6 EBS Encryption Misconfiguration / Encryption at Rest

Data-at-rest encryption misconfigurations are of particular concern in such a service as Amazon Elastic Block Store (EBS). EBS volumes are not encrypted by default, as pointed out by Guffey and Li [1], and this fact can be unexpected when the developers are new to the encryption configuration of AWS. Without encryption, sensitive information stored on these volumes may be exposed to unauthorized access or information on systems and organizations is seriously at risk.

Verdet et al. [14] list "encryption at rest" as one of the least adopted IaC security policy categories, even though it is greatly encouraged by the cloud providers. For instance, only 1.89% of AWS ECR repositories and 5.05% of instance launch configurations in the study had encryption at rest. GCP, by contrast, implements these policies by default for most services and so there is less risk associated with GCP in this context. While no particular data breach is directly linked to EBS encryption misconfiguration, its significance is highlighted throughout research [1], [14].

2.1.7 Encryption in Transit

Data encryption while in transit is another critical security requirement. Verdet et al. [14] use encryption in transit as a formal policy category in their evaluation of IaC security practices across AWS, Azure and GCP. Insufficient encryption of network communications may lead to data interception and compromise.

Paul et al. [18] further call out that such protections are often forgotten about in continuous integration and deployment (CI/CD) pipelines. Notably, GCP defaults to encryption in transit for many services, and as such, AWS and Azure require manual

configuration, which is often missed. No particular breaches involving this vulnerability are identified in real-world cases in the reviewed works, but its status as one of the first security requirements is always stressed.

2.1.8 Security Group Misconfiguration

Security groups -- virtual firewalls that control traffic -- are another common cause of misconfiguration. Guffey and Li [1] indicate that rules that are too permissive like open ports with access from any IP address pose serious risks as they allow attackers to directly interact with exposed services. These weaknesses are often used as entry points for more complex chains of attacks. A famous example was the Capital One breach where a misconfigured web application firewall (WAF) or security group is thought to have been the initial attack vector. Verdet et al. [14] agree that security group policies tend to be misused in IaC scripts, with widespread failure rates in basic access control enforcement. Paul et al. [18] raise similar concerns with regard to network security across CI/CD workflows more broadly.

2.1.9 General Configuration Data Issues

Another typical cause of cloud misconfigurations is security groups that serve as virtual firewalls to regulate the traffic entering or going out. Guffey and Li [1] explain that excessively lax rules, e.g. open ports where anybody can use any IP address, pose severe security threat, as they allow attackers to communicate directly with exposed services. These vulnerabilities are commonly employed as initial stages of more complicated attacks. One such famous incident is the Capital one data breach, in which a misconfigured web application firewall (WAF) or security group is suspect to have been used as the first point of attack by the attacker.

2.1.10 Logging and Monitoring Issues

The last big category is the failure to carry out proper logging and monitoring. Verdet et al. [14] conclude that this category of policy is inconsistently adopted, especially in AWS and Azure IaC configurations. For example, only 15.89% of evaluated templates allowed for VPC Flow Logs, and only 11.77% had access logs for S3 Bucket.

These omissions impair the ability to identify anomalies or effectively respond to incidents.

Paul et al. [18] report similar deficiencies in CI/CD pipeline configurations: Lack of sufficient monitoring is a recurring problem. Although GCP is better at promoting the adoption of log, there are still significant gaps between providers.

2.2 Real-World Consequences of Cloud Misconfigurations

The consequences of cloud misconfigurations are far-reaching and severe and can often lead to data breaches, unauthorized access, service outages, and the compromise of entire infrastructures. As organizations continue to rely on cloud services and Infrastructure as Code (IaC) for managing deployments, the dangers of having the wrong configurations have grown in magnitude and frequency. Several real-world incidences and studies demonstrate the important risks of these mistakes.

2.2.1 Data Breaches and Data Leakage

Data breaches remain a major threat in the cloud environment, and misconfigurations are one of the main causes. Madnick [19] points out that misconfigured cloud storage, permissive access settings and unsecured cloud resources make for easy pickings for attackers. As more companies are moving their data to the cloud, these configuration errors become even more critical and consequently, result in substantial exposure of data.

An example is the 2023 MOVEit supply chain attack that affected over 2,600 companies across the world exposing the personal data of millions of individuals. Equally, in 2019 the Capital One data breach took advantage of the misconfiguration, or how an attacker can access with the help of a weak Web application firewall (WAF) and thereafter exploit weak IAM settings to access sensitive data stored in S3 buckets. This breach, involving over 100 million users, shows how serious the misconfigurations of the clouds can be.

Madnick [19] also refers to the growing threat of ransomware-as-a-service and of exploiting vendor systems, which add to the threats of cloud misconfigurations. These

factors make it more difficult than ever to ensure the security of cloud environments, especially when over 80% of data breaches in 2023 involved cloud-based data, often because of improper configuration and lack of oversight.

Such configurations of cloud storage as publicly accessible or open S3 buckets remain a significant source of security breaches. The fact that attackers can easily find and abuse these open cloud resources points to the necessity of more intense security practices. The NSA research reports that misconfiguration is also among the most prevalent vulnerabilities in a cloud environment. Such mistakes are common as they are either ignored or misinterpreted, in particular, by those people and companies which do not have enough experience of using cloud technologies.

2.2.2 Malicious Code Injection

Misconfigurations can also be used to allow for the injection of malicious code into cloud systems. When access tokens or credentials are not appropriately secured, such as is the case with badly configured AWS Cognito setups, attackers can use these vectors to access backend services. Guffey and Li [1] point this out as a risk especially in the case of misconfigurations which expose APIs or interfaces to unauthorized manipulation.

The Twilio SDK attack in 2020 is a very notable example. In this incident, attackers got unauthorized access to cloud storage and could inject malicious code into Twilio's software development kit (SDK). This not only affected data confidentiality, but also effected the integrity of the software product that was delivered to downstream users [1].

2.2.3 Service Disruption and Outage

Service availability can be seriously impacted by misconfigurations, even if there is no malicious activity. Cloud services rely upon proper configuration for their reliability and performance. Guffey and Li [1] link configuration errors to serious disturbances of availability, including catastrophic outages.

The Facebook cloud outage in October 2021 provides a good example of this outcome. A single misconfig in the routing infrastructure caused a six-hour outage across

Facebook, WhatsApp, and Instagram to 2.9 billion users worldwide. While the misconfiguration was not the result of a security breach, its consequences were global in scope and illustrate how fragile cloud operations can be when vulnerable to simple errors when setting things up [1].

In addition, Dai et al. [4] demonstrate that risky scripts in IaC deployments (which often include configuration flaws) can result in high-severity system failures. These problems directly affect the availability and stability of the applications hosted in the cloud and are often hard to spot before the applications go live.

2.2.4 Unauthorized Access and Lateral Movement

Cloud misconfigurations often provide an easy path to unauthorized access and lateral movement across the cloud infrastructure. Guffey and Li [1] explain that attackers can use open S3 buckets, permissive IAM roles and open network interfaces for a foothold on the environment, and then traverse the environment, increasing privileges or deeper systems.

In the Capital One breach, the attacker could laterally navigate an entry point in a misconfigured firewall and take advantage of IAM control misconfigurations and ultimately gain access to S3 storage. Otherwise, the breach of Twilio commenced with a misconfiguration of cloud storage, which then expanded to large-scale data exposure [1]. These examples demonstrate that one wrong configuration choice in the cloud can cause a far greater security breach.

These events illustrate the way in which the cascading misconfigurations create a chain reaction and amplify the threat beyond the initial point of compromise.

2.2.5 Compromise of Entire Cloud Infrastructure

In the worst-case scenarios, a combination of multiple misconfigurations can pile up, eventually resulting in the total compromise of a cloud environment. Guffey and Li [1] point out that incorrect or insufficiently strong IAM policies, particularly those which permit privilege escalation, can provide an attacker with almost unfettered control over a

victim's cloud infrastructure. When these types of vulnerabilities are chained, it allows an attacker to control the entire compute resources, database, storage systems, and other important services.

The danger is even greater when it comes to large-scale deployments, where this configuration sprawl is hard to track and correct manually. In such environments, IaC workflows can inadvertently spread insecure configurations across hundreds or thousands of resources, causing an oversight to become a widespread security weakness.

2.2.6 Hindered Reliability of Automated Deployment Pipelines

Misconfigurations in IaC scripts can cause disruptions in automated deployment pipelines, which can result in inconsistent or failed deployments. Rahman and Williams [8] examine the issue of defective IaC scripts and show how faulty configurations can make reliability measures difficult.

Although not always considered direct security threats, the predictability and correctness of cloud service delivery is affected by these errors. In one instance described by Rahman and Williams [8], an improperly configured IaC script caused the unintentional deletion of home directories for around 270 users in the Wikimedia Commons cloud environment - a case which demonstrates the high-stake consequences of infrastructure errors.

Dai et al. [4] also reinforce this point made by automatically identifying risky deployment scripts that can produce similar failures in production environments.

2.2.7 Non-Compliance with Regulatory Standards

Cloud misconfigurations can also result in the violation of data protection laws and regulatory standards. Verdet et al. [14] found that the implementation of security policies in IaC is uneven between cloud providers, resulting in the gaps in compliance with standards like GDPR, HIPAA, PIPEDA, and CCPA.

Paul et al. [18] further point out that misconfigured CI/CD pipelines are often not configured with the logging, monitoring, and access control, which are required for

regulatory compliance. These deficiencies not only put organizations at risk of legal penalties and damage to their reputation, but also the technical risks already discussed.

2.3 Cloud Providers and Services Most Vulnerable to Misconfigurations

Cloud misconfiguration happens across all of the major cloud providers, but the frequency and impact depend upon the way that practitioners implement security policies, and how providers are structured and the defaults they apply to their services. While there is no source available, which explicitly states that one cloud provider is more vulnerable than another, comparative studies on Infrastructure as Code (IaC) practices and real-world breach studies provide information on platform-specific trends of misconfigurations and risk exposure.

2.3.1 Comparative Misconfiguration Trends Across Providers

Madnick [19] underscores the fact that misconfigurations in the cloud are one of the most common causes of data breaches, and misconfigurations of an S3 bucket and IAM policies are frequently exploited. The research highlights the growing dangers of misconfigurations across cloud providers, with the mistakes becoming the most common vulnerability in cloud environments in 2023. This trend is in line with findings by Verdet et al. [14] who studied practices related to IaC across AWS, Azure, and Google Cloud Platform (GCP) by using Terraform repositories from GitHub. Their study showed that while not every provider was inherently more vulnerable than others, AWS had the highest adoption rate of secure configurations in IaC scripts, followed by GCP and Azure. The following pass rates for adopting recommended security policies were:

- **AWS**: 76.5%
- **GCP**: 64%
- **Azure**: 58.6%

These findings suggest that **IaC practitioners** are more likely to configure **AWS** environments with greater adherence to security best practices. However, as Verdet et al.

[14] caution, these statistics are based on open-source projects and do not necessarily reflect the native security vulnerabilities of the cloud platforms themselves.

When it comes to default security settings, GCP excels with default enforcement of encryption at rest and in transit to a greater extent, which takes the pressure off of IaC security configurations to have these default policies. In contrast, AWS and Azure necessitate a more manual configuration of the encryption setup that may lead to more frequent encryption misconfigurations if the default values are not properly overridden, as pointed out by Dikshanta and Verma [20]. Their study on S3 public access detection highlights how misconfigured cloud storage and access policies are still a major challenge for both AWS and Azure. Furthermore, IBM and Ponemon [21] indicate that these types of misconfigurations have been instrumental to detect and prevent through the use of AI-driven security tools possibly in environments where the manual configuration of security settings is more susceptible to error.

2.3.2 Cloud Services Most Prone to Misconfiguration

Across the board, different cloud providers have certain services that are consistently more prone to misconfiguration - and it often has to do with their complexity, the level of access they control or the amount of manual security configuration they require. Several studies point out the following services as being especially vulnerable:

- **Cloud Storage Services:** Incorrect configuration of resources used for storage, especially Amazon S3, are among the most exploited issues. Guffey and Li [1] state that overly permissive IAM policies may offer opportunities to privilege escalation and to lateral movement from a compromised environment. The risks are even higher when the access keys are exposed or embedded in applications and thus provide an easy route for attackers to take advantage of these weak configurations.
- **Identity and Access Management (IAM):** IAM settings are misapplied and also often have much more privileges than they require. Guffey and Li [1] emphasize the fact that overly permissive IAM policies provide for privilege escalation and lateral movement in compromised environments. These problems are

compounded when access keys are left exposed or embedded within applications.

- **Networking and Firewalls:** Misconfigured network access controls such as AWS Security Groups are another major source of vulnerabilities. Allowing inbound traffic from unrestricted IP ranges (e.g. 0.0.0.0/0) has been documented as a contributing factor in the Capital One breach [1] and Paul et al. [18] cite firewall misconfigurations as recurring flaws across CI/CD deployments.

- **Cloud Compute Services:** Compute instances themselves (e.g. EC2) aren't usually misconfigured directly, but they will often be a pivot position if associated IAM roles or security groups are misconfigured. For instance, the misconfiguration of IAM on an EC2 instance had a pivotal role to play in the Capital One incident [1].

- **Other Noteworthy Services:** Service that is mentioned by Guffey and Li [1] as a service that can be misconfigured is AWS Cognito. If attackers can get access to tokens generated by Cognito, then they can interact with AWS APIs and potentially manipulate backend services.

2.3.3 Real-World Case Studies of Misconfigurations

There are multiple real-world examples of cloud security breaches, which are caused by errors in the configuration of specific services. These examples are based mainly from AWS environments as it enjoys the widest use and there is case data available:

- **Capital One Breach (2019):** This incident was caused by cascading misconfigurations. An attacker exploited a misconfiguring firewall or sec group and utilized too much IAM privileges on an ec2 instance to access sensitive s3 data. The breach affected more than 100 million people and cost the company

hundreds of millions of dollars[1].

- **Twilio SDK Attack (2020):** Attackers used misconfigured cloud storage to exfiltrate data and also inject malicious code into Twilio's software development kit. The context suggests that AWS S3 was involved although the provider is not specifically mentioned [1].

- **Imperva Breach (2019):** This breach was the result of cloud misconfigurations that allowed unauthorized access, further highlighting the risks of exposed cloud services[1].

- **Experian Breach (2017):** Identified as another example involving misconfigured S3 buckets, this breach underscores how widespread and long-standing this issue is[1].

- **Facebook Outage (2021):** Although not attributed to a particular provider, Guffey and Li [1] cite this as being a scenario where a single misconfiguration caused a massive outage that occurred globally across Facebook, WhatsApp, and Instagram, affecting 2.9 billion users.

While these examples are centered around AWS, it should be noted that neither Verdet et al. [14] nor Paul et al. [18] report similar large-scale public breaches directly related to Azure or GCP misconfigurations. However, this does not mean those platforms are inherently more secure - only that there are fewer case studies that have been made publicly available in the reviewed literature.

2.3.4 Provider-Specific Mitigation Features

Each cloud provider offers a range of built-in services designed to help mitigate misconfiguration risks:

- **Secrets Management:** AWS Secrets Manager, Azure Key Vault, and Google Cloud Secret Manager allow organizations to store and manage sensitive credentials securely [18].

- **Audit Logging and Monitoring:** AWS CloudTrail and GCP Chronicle provide centralized monitoring and logging capabilities that can detect unauthorized activity and changes to configuration settings [18].

- **Default Security Policies:** As mentioned by Verdet et al. [14] note that GCP's default enforcement of encryption policies are stronger, resulting in a higher baseline level of security which reduces the burden on developers who would otherwise need to configure these settings manually in their IaC scripts.

Despite these tools, secure configuration is the responsibility of the practitioners in large part. The same tools can be short cut or misused especially when used without validation or automation.

2.4 Security Smells in Infrastructure as Code (IaC)

Security smells in Infrastructure as Code (IaC) are recurring patterns which provide a hint of potential security weaknesses. Unlike direct vulnerabilities-such as concrete, exploitable flaws-security smells are early warning signs of poor design or insecure implementation which can become vulnerabilities if left unaddressed. This concept has been widely studied in empirical and qualitative research, most notably by Rahman et al. [3], and then developed by Chiari et al. [5] and Kumara et al. [15].

Rahman et al. [3] identified seven core security smells based on a deep analysis of Puppet scripts. A follow-up replication study extended these results to other IaC tools such as Ansible and Chef tools [5]. The following subsections provide the smells, the danger inherent in these smells, examples of these smells as seen in real-world repositories, and methods for their detection and mitigation.

2.4.1 Admin by Default

The assignment of administrative privileges to default user accounts in IaC scripts is an example of this smell. It breaches the principle of least privilege which promotes granting users the minimal permissions required for functionality [3]. Such setups result in a larger attack surface and may result in privilege escalation when exploited. Although this smell was the least common by far in datasets across the board, such as Github, Mozilla, OpenStack, and Wikimedia, it is a significant concern.

According to Rahman et al. [3] the administrator should set default accounts with the minimum privileges possible. The SLIC static analysis tool created by the authors uses pattern-based rules to detect this smell. Kumara et al. [15] also confirmed the existence of this smell in gray literature, which further validates the importance of this smell in academic and industrial settings.

2.4.2 Empty Password

This smell refers to the assigning of an empty string or weak literal (e.g. 'password') as the value for password fields. This is very closely aligned with Common Weakness Enumeration (CWE) entry CWE-259: "Use of Hard-coded Password." This makes unauthorized access ridiculous by attackers.

Though Rahman et al. [3] do not separate out the frequency of this smell, it is included in the larger "hard-coded secret" category. The SLIC tool flags attributes or variables such as password or pwd that have an empty value (string). While this particular smell was not reported in the gray literature reviewed by Kumara et al. [15], its technical meaning is obvious.

2.4.3 Hard-Coded Secret

One of the most common and least safe smells, the hard-coded secret pattern is using sensitive credentials, such as passwords, usernames, or private keys, as part of IaC scripts. This has a lot to do with CWE-798 ("Use of Hard-coded Credentials") and CWE-259.

R In their study, Rahman et al. [3] discovered 16,952 different cases of hard-coded secrets, covering four large datasets with 4 major companies, namely GitHub, Mozilla, OpenStack, and Wikimedia. These were classified as 68.3% hard coded keys, 23.9% usernames and 7.8% passwords. Worryingly, these secrets stayed in the code over long durations - the median lifetime of 20 months and the largest at 98 months - and gave an attacker sufficient time to find and utilize them.

SLIC tool identifies such patterns under the rule of a static analysis in relation to frequent string patterns and naming rules (like pwd, pass, or private key). To lessen this risk, the following mitigation measures can be recommended: isolating secrets with the help of such secure tools as Vault, or scanning credentials with the help of such tools as CredScan. The problem has also been well identified in the gray literature of industry with practitioners strongly consenting on the seriousness of the issue as well as the urgency of correcting it [15].

2.4.4 Invalid IP Address Binding

This smell is when services are set to listen on the IP address 0.0.0.0, and therefore accept connections from all interfaces. While it may make access much simpler during development, this greatly increases the exposure by enabling access from any network.

Rahman et al. [3] found this smell in a subset of the scripts that they manually reviewed, with the pattern $bind_host = '0.0.0.0' being a representative example. The SLIC tool identifies such configurations by examining variable values which resolve to this address. Although there are no specifics on mitigation strategies, standard recommendations call for limiting IP bindings to internal or limited addresses to minimize attack surfaces.

2.4.5 Suspicious Comment

This smell actually includes comments in the code that refer to known bugs, incomplete features or even potential weaknesses - using keywords like bug, fixme, todo, links to bug trackers. While this is not a direct vulnerability, such comments can reveal exploitable knowledge about the system to the attacker.

Some examples as used by Rahman et al. [3] are references to bug tracking systems in comments (e.g. # addresses bug: https://bugs.launchpad.net/ This smell had a low rate of occurrence of eight occurrences found during manual inspection. Interestingly, only 16% of surveyed practitioners judged it as a smell that requires immediate action. It was not available in the industrial gray literature reviewed by Kumara et al. [15].

Detection in SLIC is done by lexical analysis functions such as isComment(x) in combination with string-matching operations for bug-related terms (hasWrongWord(x) or hasBugInfo(x)). Although it is not a priority to be fixed for the remediation, its existence can still signal neglected or unstable code areas.

2.4.6 Use of HTTP Without TLS

This smell points out the way services are configured to use the insecure HTTP protocols rather than use the secure protocol of a web page (HTT/ which uses the TLS protocol). The use of the protocol, http, leaves data wide open to eavesdropping and tampering.

Chiari et al. [5] found that this is one of the more common smells in Ansible and Chef scripts. Rahman et al. [3] report high degree of practitioner agreement on this being a valid smell-- with 100% agreement in the Ansible replication study. Detection is done in SLIC by scanning for URL strings that begin with http:. Practitioners are encouraged to enable TLS and use available infrastructure tooling to make it easier for them to set up secure protocols.

This smell was not found by Kumara et al. [15], probably because of the widespread assumption that the use of the HTTP is already not recommended in production environments.

2.4.7 Use of Weak Cryptographic Algorithms

Weak cryptographic algorithm is used in cases whereby developers use outdated, insecure hash functions such as MD5 or SHA-1. These algorithms were popular but nowadays they can be broken easily by attackers through the collision attacks or preimage

attacks. It implies that an attacker may develop another input that may result in the same hash or even attempt to backtrack the hash to obtain the original data. Rahman et al. [3] discovered actual incidences of this issue, including MD5 hashes embedded directly in configuration documents. The problem was prevalent in most datasets and nearly 85% of security practitioners admitted that it indeed needs to be addressed.

SLIC scans its code to determine the presence of this smell by checking whether the functions call known weak algorithm names. When something such as MD5 or SHA-1 comes into view, then it is put as being insecure. The solution, which is recommended to be adopted by the developers is to adhere to the latest cryptographic guidelines of NIST and abandon old algorithms. Rather than that, they ought to apply more powerful and stable hashing algorithms like SHA-256 or SHA-3, which are much more secure and can help them secure their systems against the contemporary cyber-attacks..

2.5 Effectiveness of Tools for Detecting Security Smells in Infrastructure as Code

As Infrastructure as Code (IaC) becomes one of the cornerstones of modern cloud infrastructure management, the ability to automatically detect security risks has become a must. IaC scripts help us to quickly provision and scale up the cloud resources but also they bring the software engineering issue such as configuration defects and security smell that can compromise the deployed environment. Tools that analyze IaC code for insecure patterns contribute in a crucial way to mitigate these risks, especially by static analysis techniques.

This section reviews the effectiveness of some of the key tools identified in the literature - namely SLIC and SecureCode, as well as supporting tools, such as Checkov, SLAC, and others. These tools are designed to detect misconfigurations and security smells at an early stage in the development lifecycle, allowing for proactive mitigation.

2.5.1 SLIC: Security Linter for Infrastructure as Code

SLIC is a static analysis tool that aims at finding specific security smells in Puppet scripts. Developed by Rahman et al. [3], it identifies insecure coding patterns

systematically that are indicative of potential security flaws. SLIC works by parsing IaC scripts and extracting tokens such as keywords, variables, and values from those scripts. It then performs rule-based analysis based on the Puppet-Lint API, matching the patterns detected with the known Common Weakness Enumerations (CWEs).

- **Smells Detected by SLIC:** SLIC was designed specifically to identify the 7 smells found by Rahman et al. [3]: admin by default, empty password, hard-coded secret, invalid IP address binding, suspicious comment, use of HTTP without TLS and use of weak cryptographic algorithms. Among these, hard-coded secrets were found as the most prevalent and the most persistent across large datasets.

- **Effectiveness and Evaluation:** SLIC was evaluated on a dataset of 15,232 scripts in Puppet from 293 repositories, such as Mozilla, OpenStack, Wikimedia, and GitHub. It counted 21,201 instances of the seven smells defined in the study. A controlled test on 250 scripts picked at random showed precision of 0.99, recall of 0.96, some false positives, and false negatives - mostly in detection of hard coded secrets and IP binding issues [3].

- SLIC's applicability to the real world was exhibited in 1000 automatic bug reports based on smells detected and submitted to project maintainers. Out of 212 replies, 148 reports were accepted and acted on by developers demonstrating practical relevance and acceptance of the tool's results [3].

- **Limitations:** Despite its accuracy, SLIC's power relies on the power of the underlying Puppet-Lint API, which might misclassify variables or attributes. Furthermore, its rule derivation was founded on the judgment of the authors, but with support from the verification of independent raters. SLIC is also only available for Puppet scripts; the support of other IaC platforms is a future direction.

2.5.2 SecureCode

SecureCode, introduced by Dai et al. [4], is aimed at Ansible playbooks with embedded Shell or PowerShell code. Unlike tools that deal exclusively with IaC structure or syntax, SecureCode concentrates on identifying dangerous operational behaviors in embedded scripts by composing and analyzing the scripts using tools such as ShellCheck and PSScriptAnalyzer.

- **Approach and Functionality:** SecureCode takes embedded scripts from Ansible playbooks and substitutes variables for real values and executes the resulting scripts using standard static analysis tools. It maps identified problems to possible effects on business operations, classifying them as security, availability, performance, and reliability risks.

- **Effectiveness and Evaluation:** In its evaluation on 45 IBM repositories containing 1492 automation files, SecureCode found 3535 issues, out of which 3419 were confirmed to be true positives. Approximately 1,691 issues were classified as high severity. SecureCode had better performance against other baseline tools such as ShellCheck, PSScriptAnalyzer, and Ansible-lint [4] in terms of identifying contextually meaningful security concerns.

- **Limitations:** SecureCode's use of external analysis tools inevitably creates false positives. 116 in the study. Core Ansible code is not the main area for analysis, as it uses embedded scripts. This may restrict its ability to see a broader configuration context. In addition, SecureCode does not currently cover IaC tools other than Ansible.

2.5.3 SLAC: Security Linter for Ansible and Chef

SLAC extends the methodology pioneered by SLIC to **Ansible and Chef** scripts. As described by Chiari et al. [5], SLAC includes detection of the original seven smells and introduces two new ones: **"no integrity check"** and **"missing default case statement"**. SLAC incorporates rule-based detection and machine learning models such

as Random Forest, showing improved accuracy in smell classification compared to traditional pattern-matching approaches.

Though experimental, SLAC represents a step toward generalizing smell detection across multiple IaC frameworks. It also emphasizes scalability by enabling smell detection in CI/CD environments.

2.6 Impact of Security Smells on IaC Security

Infrastructure as Code (IaC) security smells are repeated features of configuration and deployment scripts that provide suggestions of vulnerabilities in cloud configurations. Although these smells are not the actual vulnerabilities per se, they tend to foster the environment where the actual security issues may arise. Consequently, they are able to undermine the overall security pose of a system, diminish its reliability, and make cloud infrastructure more difficult to maintain. In this section, the direct and indirect effects of these smells are brought out as is explained in available literature.

2.6.1 Security Risk and Vulnerability Introduction

Security smells are precursors to vulnerabilities that reveal poor coding and configuration practices which might be exploited by the attacker. Rahman et al. [3] define smells, such as admin by default, empty password, hard-coded secret, invalid IP address binding, suspicious comment, use of the http without tls and use of weak cryptography algorithms, as indicators of insecure practices that weaken the security integrity of IaC scripts.

For example, giving administrative privileges to default users (admin by default) is the opposite of the principle of least privilege and adds to the attack surface of the system [3]. Similarly, hard coded secrets - credentials put directly into the code - can be stolen and abused by attackers, particularly if they are stored in public repositories. The use of insecure defaults like binding to 0.0.0.0 or using weak cryptographic functions (e.g. MD5, SHA1) leave the infrastructure to external threats [3], [5]. According to Paul et al. [18], these smells can play a role in misconfigurations that support privilege escalations, data leakage or unauthorized access.

2.6.2 Real-World Consequences and Failures

Smells encoded in IaC scripts can result in real-world disruption and security breaches. Dai et al. [4] note that risky coding patterns in embedded scripts in IaC files can cause reliability problems across the system or even catastrophic failure. When IaC is used to provision systems at scale, a single flawed construct (not coded insecurely) can spread flawed configurations throughout the infrastructure.

Empirical studies link security smells with events such as the GitHub outage, caused by an improper configuration of IaC [16] and the erasure of user home directories in Wikimedia Commons caused by an IaC error and resulting in the loss of roughly 270 user environments [8]. Guffey and Li [1] associate misconfigurations associated with security smells with severe breaches, such as the Capital One and Imperva data exposures, where improper IAM roles, insecure keys and misconfigured storage played a role in the exploitation.

2.6.3 Persistence of Smells in Codebases

The security smells can persist is a major concern, since long lived smells provide more time and opportunities to the attackers to exploit. Rahman et al. [3] state that hard coded secrets persist in scripts for up to 98 months, with a median lifetime of 20 months, and hence provide ample exposure risk. This longevity makes it critical to have proactive and continuous analysis of IaC scripts throughout development and maintenance cycles.

2.6.4 Frequency and Severity in Practice

The ubiquity of these smells can be seen from the data. Rahman et al. [3] found 21201 instances of security smells in 15232 Puppet scripts downloaded from open source repositories such as GitHub, Mozilla, OpenStack and Wikimedia. Of note, 1,326 of these were hard coded passwords, and make this smell the most common, and most severe in terms of occurrence and density - from 1.5 to 2.1 occurrences per 1,000 lines of code.

Practitioner feedback is also indicative of the significance of dealing with these smells. When 1,000 occurrences of smells have been reported to developers, 69.8% were recognized as valid bugs, especially high agreement on smells such as use of weak

cryptographic algorithms (84.6%) and use of HTTP without TLS (>75%) [3], [5]. Dai et al. [4] further report that out of 3,419 confirmed issues found by their SecureCode tool in 45 IBM repositories, 1,691 were considered high severity, confirming the operational impact of these patterns.

2.6.5 Hindrance to Reliability and Maintainability

Beyond security, smells have a negative impact on the reliability, availability and overall quality of IaC deployments. SecureCode's classification model splits problems into categories such as availability, performance, and reliability as well as security [4]. Chiari et al. [5] also link security smells with lower code quality and a higher chance of bugs.

Poorly written IaC scripts with ordering violations or missing notifiers can cause a break in the automation pipeline and can result in infrastructure that is unreliable or hard to debug [4]. Rahman and Williams [8] confirm that such flaws not only cause runtime problems but can also destroy confidence in automated deployment systems.

In addition, Borovits et al. [7] demonstrate that linguistic inconsistencies, while not categorically considered security smells, have a negative impact on the readability and maintainability of the code. This has the indirect effect of compounding security risks as unclear or inconsistent code is more likely to have persistent misconfigurations or incorrect updates.

2.7 More Tools for Detecting Security Smells and Misconfigurations in Infrastructure as Code

Detecting security smells and misconfigurations in Infrastructure as Code (IaC) are critical to the security and reliability of the cloud infrastructure deployments. Various static analysis tools and frameworks have been created to help automate the process of finding insecure patterns, ensure adherence to cloud security best practices, and minimize the risk of introducing vulnerabilities through deployment. This part provides a detailed analysis of some of the notable tools that were evaluated in the literature based on the

strictly referred studies in terms of their main functionalities, detection capabilities, empirical performance, and limitations.

2.7.1 Checkov

Checkov is an open-source static analysis tool used to identify security misconfigurations in Infrastructure as Code (IaC) scripts. Its main use case is the analysis of Terraform configuration files where it statically scans resource definitions to check if it meets defined security policies. Checkov integrates more than 2500 built-in policies that enforce security and compliance standards based on industry-recognized benchmarks, including CIS AWS Foundations Benchmark, AWS Foundational Security Best Practices, CIS Microsoft Azure Foundations Benchmark and CIS Google Cloud Platform Foundation Benchmark. One of its distinguishing features is its graph-based scanning capability which allows the tool to reason about configurations in deployment context and analyze relationships among multiple resources simultaneously, making it suitable for finding more complex misconfigurations than single-parameter rule violations[(14)] .

The types of issues Checkov detects are misconfigurations in respect of Access Policies, overly permissive default administrative rights ("Admin by default"), insufficient Encryption at Rest or in Transit, presence of Hard-Coded Secrets, broad IP Address Binding, inadequate Logging and Monitoring, and use of Outdated Features. These categories are similar to those used for classifying security policies in the empirical study conducted by Verdet et al. [14].

In that study, Checkov was used to analyze 812 open-source repositories, which were hosted on GitHub and contained Terraform configurations that targeted three major cloud platforms: AWS, Azure and Google Cloud Platform (GCP). A total of 59,045 checks were run for AWS repositories, 5,385 for Azure and 23,127 for GCP. The results showed different pass rates across providers - they have shown a pass rate of 60.8%, 56.27% and 54.97% for AWS configurations, GCP and Azure respectively. Access Policy and IP Address Binding were the best-implemented policy categories among all three cloud platforms while Encryption at Rest was the worst implemented. The study mentioned that AWS and Azure projects had improved with policies regarding

Encryption in Transit and Steering Clear of Hard-Coded Secrets, and GCP projects tend to not enforce these policies. On the other hand, Logging/Monitoring and Outdated Feature policies were more likely to be neglected in AWS and Azure, while GCP projects neglected Encryption and Secret management more often [14].

Checkov was chosen over other tools like tfsec, terrascan, and semgrep because of its greater compatibility across different IaC languages and cloud providers. However, despite its comprehensive policy base and extensive adoption, not all of the built-in policies may be applicable to every project context, and the tool is prone to false positives and false negatives, the authors said. Nevertheless, Checkov's popularity and wide utilisation make the authors assume that such inaccuracies are not heavily biasing aggregate results. Madnick [19] also emphasizes the critical role of AI-powered solutions such as Checkov in helping to combat cloud misconfigurations and mitigate breach risks as cloud misconfigurations cause over 80% of data breaches. Meanwhile, IBM and Ponemon [21] focus on the importance of identifying vulnerabilities early in the deployment process using tools such as Checkov to avoid breaches with significant costs, especially when combining it with automated detection techniques.

2.7.2 SecureCode

SecureCode is an advanced static analysis framework that resolves security and reliability problems in IaC by targeting shell scripts embedded in or referenced by Ansible playbooks. It is designed to identify risky patterns of behaviour in these scripts, in particular those that may affect availability, reliability, performance or security of the infrastructure. SecureCode works by detecting embedded shell or PowerShell scripts in IaC files, by instantiating script templates using Ansible variable values to generate concrete executable scripts, and then by analyzing them using ShellCheck (for shell scripts) and PSScriptAnalyzer (for PowerShell scripts) [4], [5].

In contrast to the standard Ansible linters that mainly focus on syntax and formatting, the strength of SecureCode is the ability to reveal a greater depth of risk by correlating the issues identified by the underlying analyzers to known security and reliability issues. It classifies the identified issues based on severity and possible impact in four fields, security, availability, performance and reliability.

SecureCode was tested in an empirical study of 45 IBM Services community repositories that contain a total of 1492 automation files. The tool identified 3535 issues out of which 3419 issues were confirmed to be valid after manual inspection, out of which 1691 were classified as high severity [4]. Another research highlights the effectiveness of the tool by comparing the results of the tool against standalone tools (ShellCheck, PSScriptAnalyzer, Ansible-lint) and demonstrating that SecureCode could find additional high-risk patterns that were not found by the baseline tools [5].

In spite of its effectiveness, SecureCode also had 116 false positives in evaluation. These inaccuracies have been attributed to either pattern-matching errors in ShellCheck and PSScriptAnalyzer or inaccuracies during the composition of executable scripts from Ansible templates. Nevertheless, its incorporation into DevOps pipelines and its execution in a production IBM cloud environment all further its practicality and real-world relevance [4], [5].

2.7.3 SLIC (Security Linter for Infrastructure as Code Scripts)

SLIC is a domain-specific static analysis tool that has been developed to find security smells in Puppet-based IaC scripts. The tool focuses on recurring coding patterns which, although they are not exploitable vulnerabilities per se, signal poor security hygiene, and potential exposure to risk. SLIC applies rule-based analysis to detect seven types of security smells: Admin by Default, Empty Password, Hard-Coded Secret, Invalid IP Address Binding, Suspicious Comment, Use of HTTP without TLS and Use of Weak Cryptography Algorithms [3].

Rahman et al. evaluated SLIC by applying it to 15,232 Puppet scripts that were collected from 293 open-source repositories from four sources: GitHub, Mozilla, Openstack, and Wikimedia Commons. The tool discovered a total of 21,201 instances of security smells, and "Hard-Coded Secret" was the most common security smell both in terms of quantity and density (occurrences per 1,000 lines of code). The share of scripts containing at least one smell was different among the repos - 17.9% in Mozilla, 26.7% in Wikimedia, 29.3% in GitHub and 32.9% in Openstack - showing the widespread presence of such smells in production codebases [3].

SLIC's detection capacity was empirically validated by submitting 1,000 randomly selected reports of smells to project maintainers. A total of 212 responses were received and 148 of these smells were accepted and fixed by developers indicating high agreement with SLIC's findings. The tool is implemented on the back of the Puppet-Lint APIs and is able to process multiple scripts at once and produce structured reports on the output with file names, line numbers, and smell types. Although initial evaluations of the tool indicated high levels of both precision and recall, the authors note that possible future improvements include semantic analysis and combining suggestions on remediation to help practitioners resolve the detected issues [3].

2.7.4 Additional Tools for IaC Static Analysis

Several other tools are mentioned in the reviewed literature as being part of the broader landscape of IaC analysis:

- **tfsec, terrascan, semgrep:** These tools provide the static analysis capabilities to detect security misconfigurations in Terraform code. In this regard, they contain misconfiguration catalogs built-in and are designed with continuous delivery pipelines in mind. While careful comparative assessments were not given, Verdet et al. stated that Checkov had wider compatibility than these alternatives, which contributed to its choice for their study [14].

- **ShellCheck and PSScriptAnalyzer:** These are basic static analysis tools for shell and power shell scripts respectively. They are the backend engines in the SecureCode framework. While they are not designed specifically for IaC, their capabilities are essential in finding low-level script errors that may propagate misconfigurations in deployment workflows [5].

- **Bandit:** Bandit is a tool that is used to find common security issues in Python code. Its role in IaC security analysis is indirect and can be used for use cases where Python scripts are included in deployment automation. However, no direct evaluation in the context of IaC was published [4].

- **Docker Linter:** Used by Schermann et al., Docker Linter identifies violations of Dockerfile best practices and is compatible with infrastructure that is managed as part of containerized IaC workflows. It aids to enforce secure and efficient configurations for container images, although detailed performance metrics were not provided [7], [15].

- **Kumara et al.'s Tool:** This ontology-based analysis tool identifies security and implementation smells in TOSCA deployment models. It is a new model of quality assurance in model-driven IaC but no empirical validation data is given in the referenced materials [7], [15].

- **Sotiropoulos et al.'s Tool:** Aimed to identify missing dependencies and notification triggers in Puppet manifests, this tool is focused on the trace analysis and fault detection. Though not developed explicitly for security analysis, which addresses critical aspects of IaC correctness that can have an indirect influence on security [6].

The reviewed tools combined to highlight the growing emphasis on automated security analysis in IaC workflows. Checkov shows great potential to accomplish cross-provider Terraform policy validation, and SecureCode offers more in-depth inspection of embedded script logic in Ansible playbooks. SLIC provides unique contributions to understanding security smells particular to Puppet. Together, these tools provide complementary coverage of the IaC security landscape and address different formats, languages and depths of analysis.

As effective as these may be, there are limitations to each tool, including possible false-positives, scope to specific IaC languages or lack of remediation support. Nevertheless, their incremental improvement and implementation into development and DevOps pipelines is crucial for reducing security risks early on in the deployment lifecycle and ensuring infrastructure compliance with security best practices.

2.8 Unsupervised Machine Learning for Anomaly Detection in Cloud Security and IaC

Anomaly detection is an essential part in the protection of cloud environments and Infrastructure as Code (IaC) systems. Given how dynamic and complex cloud infrastructure is, traditional rule-based or signature-based detection methods are often not sufficient. In order to overcome these limitations, recent studies have investigated methods of unsupervised machine learning, which do not use labeled data, but learn patterns of normal behavior in order to spot deviations. Such methods are especially beneficial in cloud environments where labeled datasets are limited or outdated. The sources reviewed in this section address a variety of unsupervised algorithms, feature extraction strategies, performance observations and limitations applicable to the problem of anomaly detection in the context of cloud security and IaC.

2.8.1 Applied Unsupervised Learning Methods

A systematic literature review published by Nasr et al. describes three main families of methodologies in unsupervised anomaly detection in cloud security: classical machine learning (28.3% of works reviewed), deep learning (19.7%), and statistical methods (23%) [10]. Within these categories, there are a number of particular unsupervised techniques that can be used.

In classical machine learning, techniques for clustering are commonly used. These include k-means and DBSCAN that group data in clusters and detect outliers based on distance metrics [10]. K-nearest neighbors (k-NN) is also mentioned, sometimes with an unsupervised context (density-based outlier detection). Local Outlier Factor (LOF) is another method which is used to determine the local deviation of a data point from its neighbors, which reveals possible anomalies [13].

One-class classification is mentioned as one of the unsupervised techniques where a decision boundary is created around 'normal' data and anything outside the boundary is marked as an anomaly. This is a particularly useful method when only examples of normal behaviour are available, as often happens in cloud environments [10].

Unsupervised techniques based on deep learning are also represented by a large face. Autoencoders that involve encoder and decoder neural network components are used to learn compressed representations of normal data. Anomalies are detected according to the reconstruction error that is usually high for the data that was not observed during the training phase [10], [13]. Recurrent Neural Networks (RNNs) and its variants, Long Short-Term Memory (LSTM) are ideal for sequential data such as time-series metrics. They predict future behavior based on past behavior and key in unexpected results as anomalous [10]. Ghaedi and Babar also point out the use of RNNs in the modeling and prediction of anomalies in time-series data in which RNNs could be suitable for temporal sequence learning even though they are known to have some issues such as gradient vanishing [7].

Self-Organizing Maps (SOMs), which map high dimensional data into lower dimension grids whilst retaining topological properties, are also discussed. Their ability to visualise clusters of data and identify anomalies is recognised in the literature [10].

Statistical techniques are another class of unsupervised techniques. These are based on building a profile of normal operations and highlighting anomalies when behavior observed deviates from this profile beyond a certain threshold. Tools such as CloudDiag and EbAT rely on such statistical profiling of cloud telemetry data and detect unexpected behaviors as anomalies [13].

2.8.2 Feature Extraction Techniques

Effective anomaly detection is dependent on effective feature selection and extraction. The reviewed works use different strategies according to the characteristic of the input data.

General feature selection is detected as an important preprocessing step. Techniques are based on classification and clustering, some are improved with anomaly detection and causality tests. For example, Jalayer et al. propose a collaborative feature selection approach with time series analysis, anomaly detection, and causality tests for better performance in intrusion detection systems using CSE-CIC-IDS2018 dataset [11].

Time series data (typical for network and system performance metrics) is often handled under the umbrella of wavelet transforms or processed directly by RNN-based models [15]. Log data is another important data source, which is used for identification of the irregular system behavior [11]. Network traffic data, including packet-level and flow-level statistics, is extensively used in the various studies, where features are extracted from the payload analysis and traffic patterns [7], [11], [13].

System call identifiers are used to extract frequency-based features, particularly in the detection of malicious behavior at the OS level in servers on the cloud [10]. Metric distributions are another domain of features, useful in the case of finding anomalies in utility cloud systems. Autoencoders are specifically mentioned to work with variable-length data sequences and enable the extraction of reliable and compact feature representations [10].

Word embeddings are presented in some of the studies in order to represent task names and script bodies as numerical feature vectors. For example, Ghaedi and Babar use such a technique to identify linguistic inconsistencies in IaC scripts though their application is more in terms of supervised classification than anomaly detection [7].

2.8.3 Performance of Unsupervised Methods

The reviewed sources present both qualitative and quantitative evaluations of performance, although specific evaluations for each unsupervised method are not always provided in detail.

A recurring performance goal in studies is to achieve high detection rates for anomalies, and false positive rates. This is especially important as too many false alarms can make a system impractical for deployment in the real world [10]. Statistical methods are characterised for their capacity to detect recent anomalies quite reliably, but their success may depend on the quality of historical data used to build profiles of behaviour [13]. Machine learning techniques, including unsupervised ones, are acclaimed as being capable of improving with time by acquiring knowledge from observed patterns in data [13].

Quantitative evaluations are lacking for methods such as autoencoders or SOMs alone, although occasionally the bigger picture of system performance is described. For instance, Jalayer et al. prove that their approach to feature selection not only leads to higher forecast accuracy, but also a significant decrease in the model complexity, training time (by 85%), prediction time (by 15%) and cross validation time (by 97%) on the CSE-CIC-IDS2018 dataset [11]. Although these improvements are mainly associated with preprocessing and feature selection steps rather than the anomaly detection algorithm itself, these improvements highlight the importance of efficient data representation.

In contrast, research works based on CNN-LSTM architectures provide high accuracy values (98.69% and 98.30% on the CSE-CIC-IDS2018 and CSE-CIC-IDS2017 datasets, respectively) [12]. However, these results probably apply to the supervised classification of known attack types rather than truly unsupervised detection of new anomalies.

2.8.4 Challenges, Limitations, and Advantages

Various issues with deploying unsupervised anomaly detection as cloud security are found in the literature. To begin with, cloud traffic is dynamic and heterogeneous thus making it difficult to determine normal and abnormal behavior. The mass and dynamic infrastructure also presents a challenge in the implementation of the static methods of detection [13].

One major constraint is the lack of labeled datasets, which encourages unsupervised learning use with the disadvantage of benchmarking and validation. Moreover, certain techniques have a high false positive rate that makes it invalid in practice [10], [13]. No, deep learning models are powerful, and they are usually not interpretable and need a lot of tuning. In the case of RNNs, some of the common problems encountered during the process of learning include the disappearance or expansion of gradients [10]. The process of extracting features of raw data is an open problem, particularly when input space dimension is large [11].

However, unmonitored approaches have a number of benefits. They are intrinsically applicable to the situations with no labeled data and can detect the anomalies

that have never been seen before or are novel. No prior training is needed and statistical methods can be used to identify emerging trends in real time traffic. Equally, machine learning and data mining solutions exhibit flexibility and self-enhancement with time and are thus appealing to changing cloud-based requirements [13].

2.8.5 Specific Implementations and Evaluations

A number of examples are used to explain the operationalization of unsupervised methods. Some of the clustering methods are mentioned in reference such as k-means, DBSCAN and LOF and are used in various implementations in cloud and network security research [10], [13]. Autoencoders are used to reconstruct patterns of inputs, as well as to identify anomalies in terms of large reconstruction errors [13]. RNNs and LSTMs are applied in predicting future behavior on a time-series data to identify anomalous events with low probability [13]. SOMs are used in dimensionality reduction and visualization of the abnormal behavior [13]. Statistical systems such as CloudDiag and EbAT use statistical finds of abnormal measures of operation to detect unforeseen activities [13].

Besides, the Anomaly Detection Tool Kit (ADTK) is applied together with feature selection in the Jalayer et al. work that discovered abnormal behavior of time series, which highlights the combination of statistical and machine learning methods in modern detection systems [11].

2.9 Summary

Cloud misconfiguration is cited by researchers as the top reason why breaches occur due to excessive permission granted to IAM roles, exposing secret keys, creating insecure network bindings and allowing unauthorized access to S3 buckets, using inadequate encryption and/or insufficient logging. The majority of these errors are reproduced via IaC, which creates a risk where a single error can be reproduced across all deployments. Past studies have identified “security smells” in IaC scripts; examples include hardcoded credentials, incorrect IP binding, use of weak crypto, and “admin by default,” which typically indicate larger security issues. There are several real world

examples of how IaC misconfigurations can cause large-scale outages: Capital One, Twilio, Imperva and Facebook's outage.

There are several static analysis tools available for detecting potential issues caused by IaC misconfigurations. Examples of some of these tools include: Checkov for Terraform, SLIC for Puppet, SecureCode for Ansible, and SLAC for support of multiple languages. While useful, these types of tools rely heavily upon manually created rule sets, produce false positives, and are generally unable to generalize across different programming languages, nor do they understand the context of the code. Research has recently moved towards utilizing unsupervised machine learning techniques such as Isolation Forests, Autoencoders, cluster algorithms and Self Organizing Maps to identify anomalous behaviors without prior knowledge of what constitutes an anomaly and adapting to changing patterns of behavior. These emerging technologies are promising for IaC analysis, however, there are still many open questions regarding feature extraction, modeling dynamic behavior of clouds, interpreting results of models, and validating results. Therefore, the development of a robust, ML-based anomaly detection tool for IaC remains an active area of research.

CHAPTER 3

METHODOLOGY

3.1 Overview of the Proposed System

This study has suggested an automated framework on anomaly detection to identify misconfigurations in a script of Infrastructure-as-Code (IaC) through an unsupervised machine learning approach. Fig. 1 shows that the system overall is divided into five main parts (1) data collection and organization, (2) Terraform parsing and tokenization, (3) feature extraction and engineering, (4) unsupervised model training and anomaly detection, and (5) visualization and explainability generation. It follows a piping execution as raw Terraform configuration files are converted to structured feature representations, run through various anomaly detection algorithms, and eventually generate actionable information through interpretable visualizations.

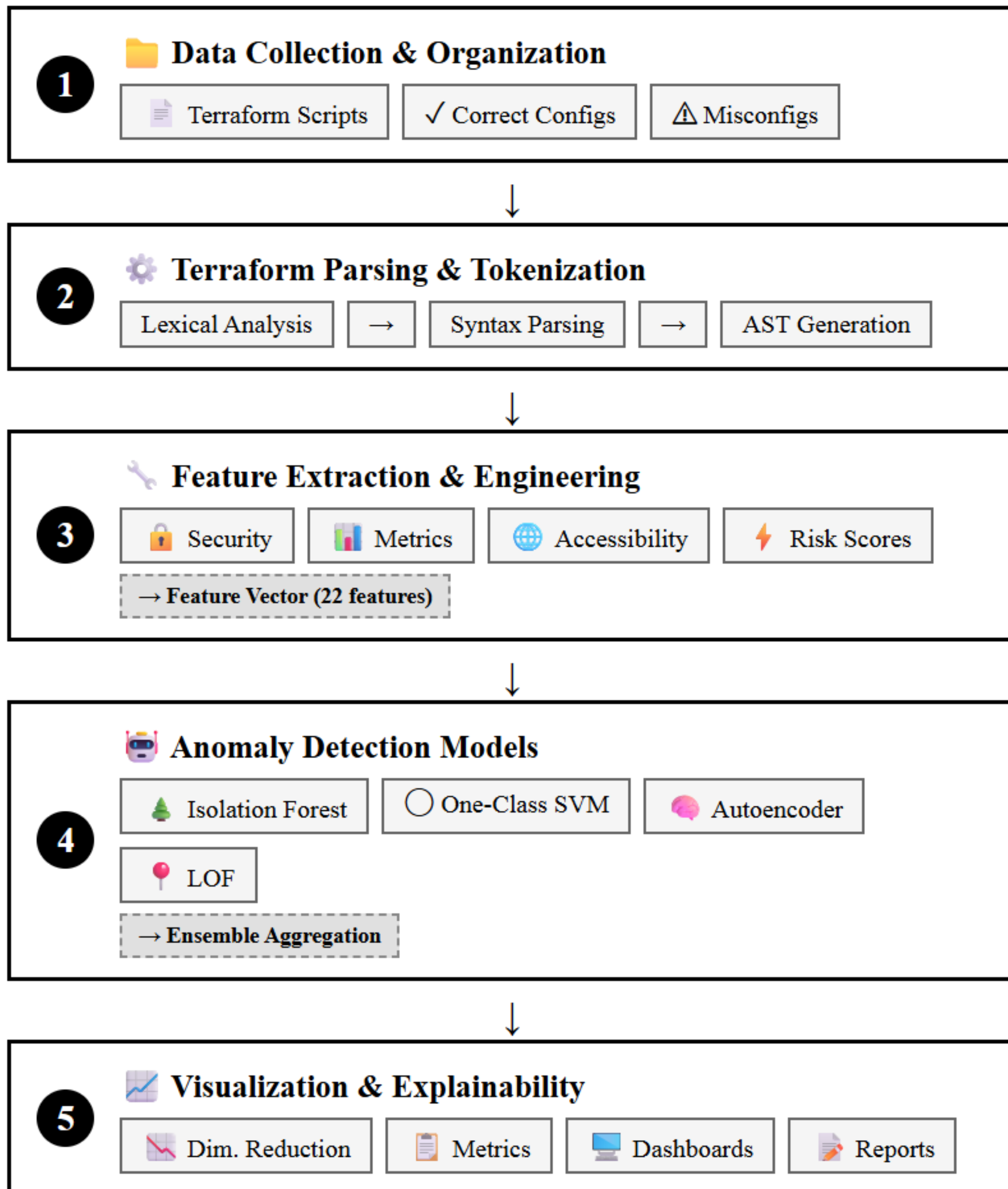


Figure 1: The overall architecture of the system with the five primary blocks.

The workflow begins with gathering Terraform .tf scripts which are further divided into two categories, the correctly configured ones (benign samples) and the ones with misconfigurations (anomalous samples). These scripts are then read very thoroughly

to extract both syntactic and semantic information. There, a complete feature engineering process is carried out to convert the information into numerical feature vectors.

These set of feature vectors are inputted into a collection of unsupervised learning models, such as Isolation Forest, One-Class SVM, and Autoencoder-based neural networks. The models compute the score of anomaly independently and the scores are added together to reach the final classification decision.

In conclusion, this system has a visualization pipeline, which generates multi-dimensional plots of the results. This enables easy interpretation and validation of the detected anomalies by the domain experts. It is all created in such a way that the whole architecture can be scaled, replicated, and easily integrated with current CI/CD processes.

3.2 Data Collection and Dataset Preparation

3.2.1 Terraform Configuration Dataset

The experimental data is the collection of Terraform configuration files gathered in open source repositories and synthetically generated examples that are intended to constitute common AWS S3 bucket misconfigurations. Let $\mathcal{D} = d_1, d_2, \dots, d_N$ denote the complete dataset, where each d_i represents a single Terraform script. The dataset is partitioned into two mutually exclusive subsets: $\mathcal{D}_{\text{correct}}$ containing 11 properly configured scripts, and $\mathcal{D}_{\text{misconfig}}$ containing 17 misconfigured scripts exhibiting security vulnerabilities. The overall number of samples is thus N=28, and the proportion between the classes is 39:61 between correct and misconfigured ones.

The misconfigurations captured in $\mathcal{D}_{\text{misconfig}}$ include but are not limited to:

1. public read/write Access Control Lists (ACLs) on S3 buckets,
2. overly permissive bucket policies allowing unauthenticated access,
3. disabled encryption at rest,
4. absence of versioning controls,
5. inadequate logging mechanisms, and
6. missing secure transport enforcement.

These patterns of vulnerabilities have been carefully introduced to reflect the security vulnerabilities that are common in the real world, that are often reported in cloud security audits, and that are listed in the Common Vulnerabilities and Exposures (CVE) databases.

3.2.2 Terraform Parsing Process

Fig. 2 shows that Terraform is a hierarchical program that decomposes.tf files, via parsing, into hierarchical program representations of Structured Abstract Syntax Tree (AST) representations. Terraform uses HashiCorp Configuration language (HCL), a block based and declarative language, which has a syntax consisting of blocks. A configuration file d i is comprised of several resource blocks whose definition is in the form of:

```
resource "resource_type" "resource_name" {
 attribute_1 = value_1
 attribute_2 = value_2
 nested_block {
   nested_attribute = nested_value
 }
}
```

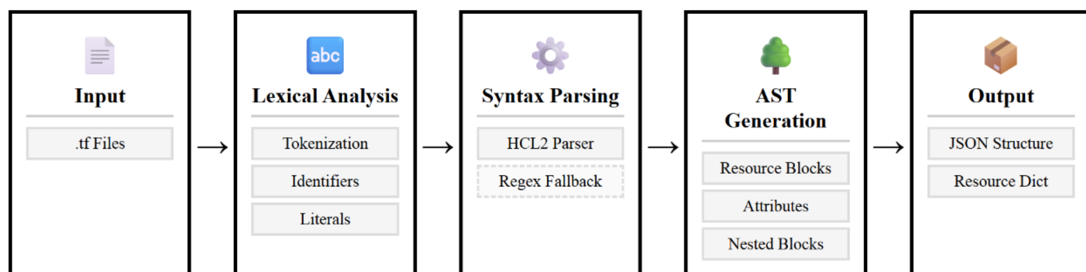


Figure 2: The Terraform parsing process.

The lexical analysis, provided by the parsing algorithm using the python-hcl2 library, is used to tokenize the HCL syntax into the basics (identifiers, literals, operators), and then syntactic analysis to build a nested dictionary representation with all the hierarchical relationships between resources, attributes, and nested blocks. Let $P(d_i)$

represent the parsing function that transforms a raw Terraform script into a structured representation:

$$P(d_i) = r_1, r_2, \dots, r_k$$

where each r_j denotes a parsed resource block with its associated attributes $A_j = a_1, a_2, \dots, a_m$.

The data types supported by the parser are as follows: strings, integers, booleans, lists and maps and the type inference and type normalization are journeyed in the parsing phase. There are error handling mechanisms to handle bad configured configurations and there are logging functions to trace parsing failures. The parsing result is produced as a serialized form of the output in a form of a JSON structure to be used further by other feature extraction functions, making it interoperable and easy to process.

3.2.3 Data Pre-processing and Normalization

Following parsing, the structured representations undergo preprocessing to standardize attribute values and handle missing data. Let $X_{\text{raw}} \in \mathbb{R}^{N \times M}$ represent the raw feature matrix extracted from parsed configurations, where N is the number of samples and M is the dimensionality of the feature space. Preprocessing operations include:

1. **Missing Value Imputation**: For boolean attributes (e.g., `versioning_enabled`, `encryption_enabled`), missing values are imputed with 0 (false) under the principle of secure-by-default. For numerical attributes, median imputation is applied:

$$x_{ij}^{\text{imputed}} = \begin{cases} x_{ij} & \text{if } x_{ij} \text{ is observed} \\ \text{median}(X_{\cdot j}) & \text{if } x_{ij} \text{ is missing} \end{cases}$$

2. **Categorical Encoding**: Categorical variables such as ACL types (`private`, `public-read`, `public-read-write`) are encoded using one-hot encoding, expanding the feature space to include binary indicator variables.
3. **Feature Scaling**: To mitigate the influence of feature magnitude differences on distance-based algorithms, robust scaling is applied:

$$x_{ij}^{\text{scaled}} = \frac{x_{ij} - Q_2(X_{\cdot j})}{Q_3(X_{\cdot j}) - Q_1(X_{\cdot j})}$$

4. where Q_1, Q_2, and Q_3 represent the first quartile, median, and third quartile, respectively. Robust scaling was preferred over standard z-score normalization due to its resilience against outliers, which are inherently present in anomaly detection scenarios.
5. **NaN Handling**: The `fix_nan_data.py` module implements systematic NaN detection and correction using `np.nan_to_num()` with replacement values: NaN → 0.0, $+\infty \rightarrow 10^6$, $-\infty \rightarrow -10^6$.

3.3 Feature Extraction and Engineering

3.3.1 Feature Definition and Extraction Strategy

The feature extraction converts Terraform configurations that are parsed into numbers that can be used with machine learning algorithms. The extracted features under four categories, security-related features, configuration complexity metrics, accessibility indicators and derived risk scores, are extracted through the feature engineering process as it is implemented in the feature engineering.py. The entire extracted feature taxonomy is represented in Table 1.

Table 1: Extracted Feature Taxonomy for IaC Analysis

Feature Name	Type	Category	Description
is_public_acl	Binary	Security	Presence of public-read or public-read-write ACL
is_private_acl	Binary	Security	Presence of private or authenticated-read ACL
has_public_policy	Binary	Security	Bucket policy allows public access
has_encryption	Binary	Security	Server-side encryption enabled
has_logging	Binary	Security	Access logging configured
has_versioning	Binary	Config	Versioning enabled
has_lifecycle	Binary	Config	Lifecycle rules defined
resource_count	Numeric	Config	Number of resources in file
total_lines	Numeric	Config	Total lines of code
public_access_blocks	Numeric	Access	Number of public access block settings
acl_grant_count	Numeric	Access	Number of ACL grants

Feature Name	Type	Category	Description
policy_statement_count	Numeric	Access	Number of policy statements
has_tags	Binary	Compliance	Resource tagging present
has_cors	Binary	Compliance	CORS configuration defined
has_website	Binary	Compliance	Website hosting enabled
security_score	Numeric	Risk	Composite security score (0-100)
accessibility_score	Numeric	Risk	Public accessibility score (0-100)
complexity_score	Numeric	Risk	Configuration complexity score
file_size	Numeric	Metadata	File size in bytes
provider_count	Numeric	Metadata	Number of provider blocks
variable_count	Numeric	Metadata	Number of variable definitions
output_count	Numeric	Metadata	Number of output blocks

3.3.2 Mathematical Feature Representation

Each Terraform configuration d_i is mapped to a feature vector $\mathbf{f}_i \in \mathbb{R}^M$, where $M = 22$ represents the feature dimensionality. The feature extraction function $\Phi: \mathcal{D} \to \mathbb{R}^M$ is defined as:

$$\boldsymbol{f_i} = \boldsymbol{\Phi}(\boldsymbol{d_i}) = [\boldsymbol{f_{i1}}, \boldsymbol{f_{i2}}, \dots, \boldsymbol{f_{iM}}]^{\boldsymbol{T}}$$

The complete feature matrix for the dataset is constructed as:

$$X = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1M} \\ f_{21} & f_{22} & \cdots & f_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ f_{N1} & f_{N2} & \cdots & f_{NM} \end{bmatrix} \in \mathbb{R}^{N \times M}$$

3.3.3 Derived Security Metrics

Three composite features are engineered to capture high-level security posture:

Security Score: Aggregates multiple security indicators into a normalized score:

$$\text{security_score}_i = \frac{1}{6} \sum_{j \in S} w_j \cdot f_{ij} \times 10$$

where $S =$ versioning,logging,encryption,secure_transport,private_acl,no_public_policy and w_j are uniform weights.

Accessibility Score: Quantifies the degree of public exposure:

$$\text{accessibility_score}_i = 8 \cdot \text{is_public_acl}_i + 2 \cdot \text{has_public_policy}_i + \epsilon$$

where $\epsilon \sim \mathcal{U}(0,2)$ introduces controlled stochasticity for realistic variance.

Risk Ratio: Measures the imbalance between accessibility and security:

$$\text{risk_ratio}_i = \frac{\text{accessibility_score}_i}{\max(0.1, \text{security_score}_i)}$$

High risk ratio values (> 1.0) indicate configurations with elevated public accessibility relative to their security controls, serving as strong anomaly indicators.

3.4 Unsupervised Anomaly Detection Models

As shown in Fig. 3, the ML model pipeline has three separate unsupervised learning models that utilize various theoretical underpinnings to detect anomalies. Ensemble technique helps reduce biases of individual models and increase their detection strength.

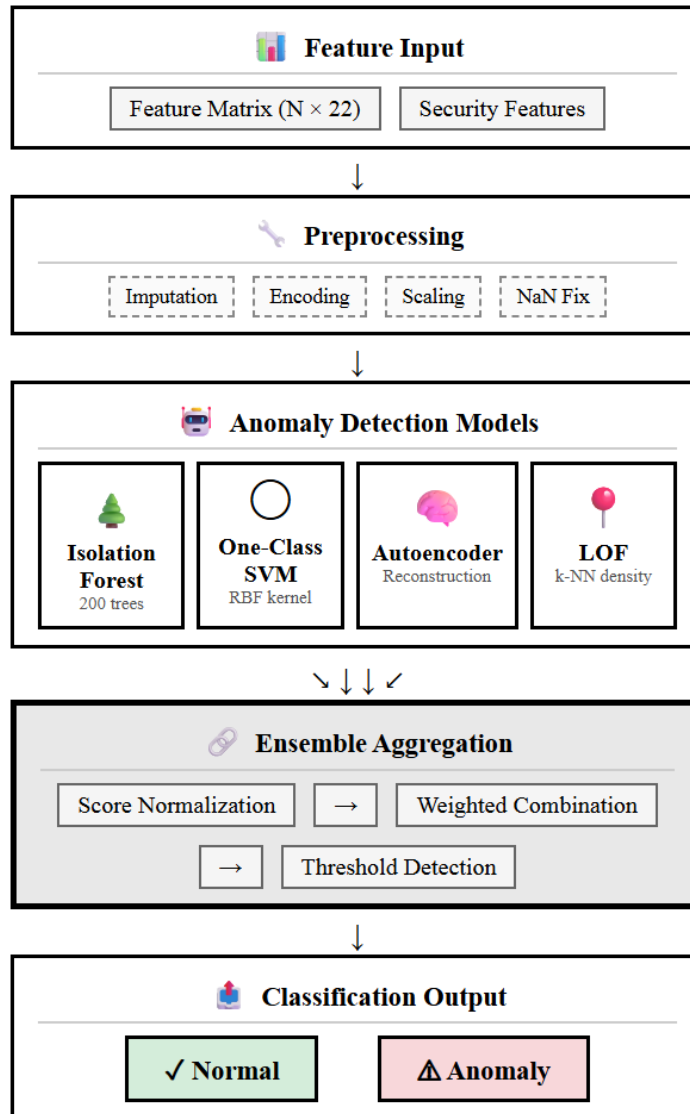


Figure 3: Machine learning model workflow.

3.4.1 Isolation Forest

Isolation Forest is based on the idea that anomalies are small and distinct, and hence are more separable in feature space. The algorithm is used to build an ensemble of isolation trees in which each tree splits the data using random selection of features, and random splitting of thresholds.

Algorithm Foundation: Algorithms Foundation: The anomaly score of a certain sample $\mathbf{x}$, is defined as the average length of its path in all the trees:

$$s(\mathbf{x}, n) = 2^{-\frac{E(h(\mathbf{x}))}{c(n)}}$$

where:

- $E(h(\mathbf{x}))$ is the average path length of $\mathbf{x}$ across all isolation trees
- $c(n) = 2H(n-1) - \frac{2(n-1)}{n}$ is the normalization factor
- $H(i)$ is the harmonic number: $H(i) = ln(i) + \gamma$ (Euler's constant)
- n is the number of training samples

The value of anomaly that is close to 1 shows that anomalies are also strong and that values close to 0 depict that there are no anomalies. The classification limit is usually fixed at $s > 0.5$, but it may be varied depending on contamination parameter, $\nu \in (0, 0.5)$.

Implementation Details: The Isolation Forest model (isolation forest.py) was set to use $T = 200$ trees, a maximum sample size of 256 in each tree and contamination parameter $\nu = 0.1$, which represents the probability of an anomaly that is expected. Bootstrap sampling was also allowed to promote diversity of trees.

3.4.2 One-Class Support Vector Machine

One-Class Support Vector Machine (SVM) creates a decision boundary which defines the distribution of normal data. Any sample that does not lie within this limit is considered to be an anomaly. The approach solves a formal optimization problem where a hyperplane is determined to maximize the distance between the training set and the source in a high-dimensional feature space which is defined by a kernel-function. This formulation allows the model to distinguish adequately regular configurations and possible outliers, and thus it can assist in detecting anomalies in complex data.

Mathematical Formulation: The primal optimization problem is:

$$\min_{w,\rho,\xi} \frac{1}{2}|w|^2 - \rho + \frac{1}{\nu n}\sum_{i=1}^{n} \xi_i$$

subject to:

$$\mathbf{w}^T \phi(\mathbf{x}_i) \geq \rho - \xi_i, \quad \xi_i \geq 0$$

where $\phi(\cdot)$ is the kernel mapping, $\mathbf{w}$ defines the separating hyperplane, ρ is the offset, $\boldsymbol{\xi}$ are slack variables, and $\nu \in (0,1)$ controls the trade-off between maximizing the margin and minimizing violations.

The decision function for a test sample $\mathbf{x}$ is:

$$f(\mathbf{x}) = \operatorname{sgn}(\mathbf{w}^T \phi(\mathbf{x}) - \rho) = \operatorname{sgn}\left(\sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}) - \rho\right)$$

where $K(\cdot,\cdot)$ is the kernel function (Radial Basis Function kernel employed in this study):

$$K(\mathbf{x}_i, \mathbf{x}_j) = exp\left(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2\right)$$

with $\gamma = \frac{1}{M}$ (automatic scaling based on feature dimensionality).

Implementation Details: One-Class SVM Model (one_class_svm.py) This model used an RBF kernel with automatic gamma computation, $\nu = 0.1$, and 500 MB for cache size on to speed computation.

3.4.3 Autoencoder Neural Network

The autoencoder method is another way to think about anomaly detection as a reconstruction problem. It learns a representation that reconstructs the normal samples that dominate the training data well. The anomalies exhibit high reconstruction error.

Architecture: The autoencoder comprises an encoder network $E: \mathbb{R}^M \to \mathbb{R}^D$ and a decoder network $D: \mathbb{R}^D \to \mathbb{R}^M$, where $D < M$ is the bottleneck dimension enforcing dimensionality reduction. The network is trained to minimize reconstruction error:

$$\mathcal{L}(\mathbf{x}) = \|\mathbf{x} - D(E(\mathbf{x}))\|^2 = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

Implementation: A three-layer architecture was implemented using scikit-learn's `MLPRegressor` as a TensorFlow-free alternative:

- Encoder layer: $M \to \lfloor 0.75M \rfloor$ neurons
- Bottleneck layer: $\lfloor 0.5M \rfloor$ neurons
- Decoder layer: $\lfloor 0.75M \rfloor \to M$ neurons

ReLU activation functions were applied at each layer. The model was trained using Adam optimizer with learning rate $\alpha = 10^{-4}$, early stopping validation (10% of training data), and maximum 500 epochs.

Anomaly Score Computation: For a test sample $\mathbf{x}$, the anomaly score is the mean squared reconstruction error:

$$s(\mathbf{x}) = \frac{1}{M} \sum_{j=1}^{M} (x_j - \hat{x}_j)^2$$

Large reconstruction error scores indicate that these samples are likely to be anomalous; they do not follow a typical/normal pattern as modelled by the algorithm.

3.4.4 Ensemble Aggregation Strategy

An aggregation strategy allows for combining the results of each of the individual models to take advantage of their respective strengths. Let $s_k(\mathbf{x})$ denote the anomaly score from model $k \in$ IF,OCSVM,AE (Isolation Forest, One-Class SVM, Autoencoder). The ensemble score is computed as a weighted combination:

$$s_{\text{ensemble}}(\mathbf{x}) = \sum_{k=1}^{K} w_k \cdot s_k^{\text{norm}}(\mathbf{x})$$

where $s_k^{\text{norm}}(\mathbf{x}) \in [0,1]$ are min-max normalized scores and $\sum_{k=1}^{K} w_k = 1$. Weights were determined through performance-based weighting, assigning higher weights to models with superior F1-scores on validation data.

3.5 Training and Validation Strategy

The dataset was split into training (70%), validation (15%), and test (15%) sets while ensuring class distributions were preserved using stratified sampling. The training set was composed solely of samples from $\mathcal{D}_{\text{correct}}$ since, in keeping with the unsupervised learning paradigm utilized in these experiments, the models learn the distribution of normal behaviour without explicitly being told about the existence of anomalies during training.

The hyperparameters for the models were tuned using a grid search over the hyperparameter space with cross-validation applied to the validation set. For Isolation Forest, this included $T \in 100,200,300$ and $\nu \in 0.05,0.1,0.15$. The choices for One-Class SVM hyperparameters included choices of kernel (RBF, linear) and $\nu \in 0.05,0.1,0.15$. and autoencoder hyperparameters were choices of hidden layer sizes and training epochs.

3.6 Anomaly Scoring and Threshold Determination

For every model, obtaining samples, we sort these samples in descending order according to their anomaly scores. The classification threshold for every model, θ, is obtained with:

$$\theta = Q_{1-\nu}\big(s(\mathbf{x})\big)$$

where $Q_{1-\nu}$ represents the $(1-\nu)$-th quantile(s) of the anomaly score distribution. Using score vs. threshold, we label samples as anomalies:

$$\hat{y}(\mathbf{x}) = \begin{cases} 1 & \text{if } s(\mathbf{x}) > \theta \\ 0 & \text{otherwise} \end{cases}$$

This adaptive thresholding infers that a set proportion of $\nu \times 100\%$ of samples should be anomalies since the domain expects a certain rate of misconfiguration.

3.7 Visualization and Explainability Pipeline

The visualization pipeline, as shown in Fig. 4, translates high-dimensional anomaly detection results into visual representations. Multiple visualization modalities were developed for result analysis.

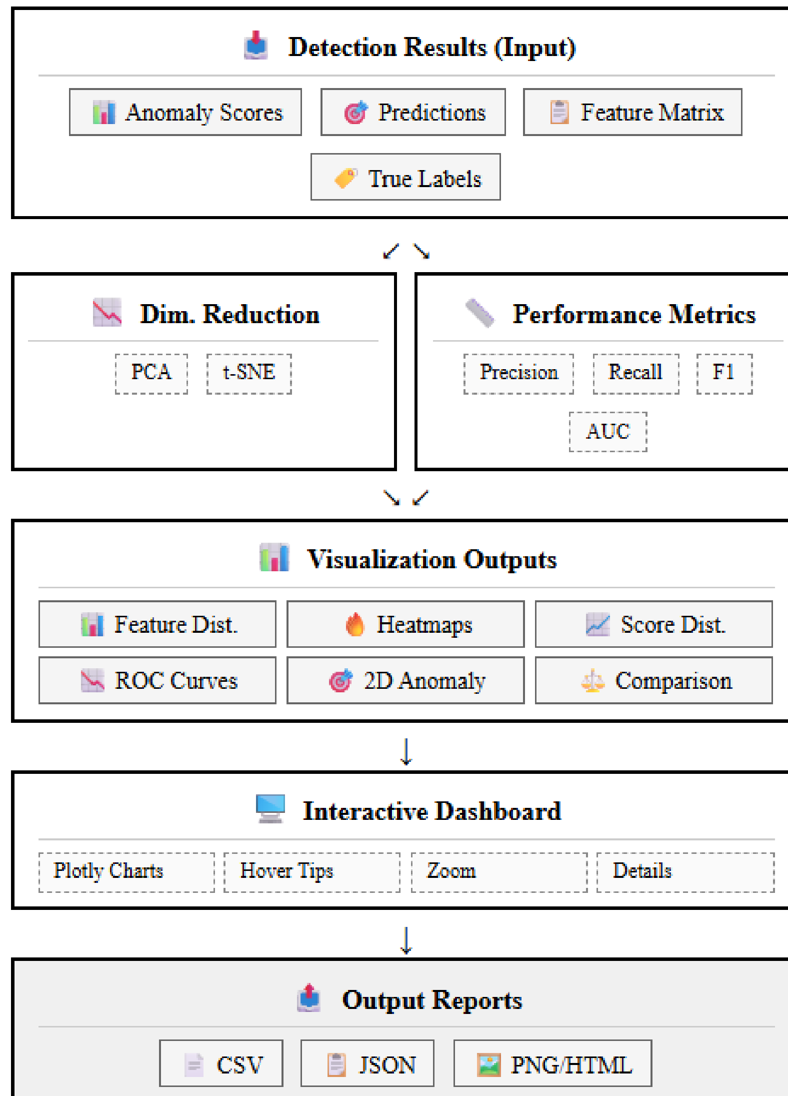


Figure 4: Visualization pipeline.

3.7.1 Dimensionality Reduction for Visualization

Principal Component Analysis (PCA): Projects the M-dimensional feature space onto a 2D subspace maximizing variance:

$$\mathbf{z}_i = W^T(\mathbf{x}_i - \boldsymbol{\mu})$$

where $W \in \mathbb{R}^{M \times 2}$ contains the top two eigenvectors of the covariance matrix $\Sigma = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T$.

t-Distributed Stochastic Neighbor Embedding (t-SNE): Performs non-linear dimensionality reduction preserving local neighborhood structure:

$$\min_{\mathbf{Y}} \mathrm{KL}(P \parallel Q) = \sum_{i \neq j} p_{ij} \, log \frac{p_{ij}}{q_{ij}}$$

where P represents pairwise similarities in the original space and Q in the embedded space.

3.7.2 Visualization Outputs

The visualization suite generates:

1. **Feature Distribution Plots**: Histograms showing distributions of key features stratified by anomaly class
2. **Correlation Heatmaps**: Visualizing inter-feature dependencies
3. **Anomaly Score Distributions**: Comparing score distributions across models
4. **ROC and Precision-Recall Curves:** Analyzing model discrimination ability
5. **2D Anomaly Detection Plots**: Displaying detected anomalies in PCA-reduced space alongside predicted anomaly overlays
6. **Model Comparison Dashboards**: Comparing performance between models' side by side.

Plotly interactive visualizations allow users to zoom into specific data points (e.g. sample, anomaly score, feature values, source file) using hover over tools.

3.8 Performance Evaluation Metrics

Although the problem is unsupervised, we evaluate the performance of the model based on the provided true labels in the test set to measure how well the model detects anomalies. We use the following evaluation methods to assess detection quality:

Precision: Measures proportion of true positives (anomalies correctly flagged by the system) to total number of flagged samples:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall (Sensitivity): Measures proportion of true positive cases that were correctly flagged:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-Score: The harmonic mean between precision and recall::

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}$$

ROC-AUC: The area under the Receiver Operating Characteristic (ROC) curve which measures a classifier's ability to discriminate between classes over all possible thresholds:

$$\text{AUC} = \int_0^1 \text{TPR}\left(\text{FPR}^{-1}(t)\right)\, dt$$

where $\text{TPR} = \frac{TP}{TP+FN}$ (True Positive Rate) and $\text{FPR} = \frac{FP}{FP+TN}$ (False Positive Rate).

Average Precision (AP): The area under a Precision-Recall Curve. AP is useful when there are highly skewed distributions (i.e., many more negative than positive examples):

$$\text{AP} = \sum_{k=1}^{n} P\,(k) \cdot \Delta R(k)$$

where $P(k)$ and $R(k)$ represent precision and recall at each k threshold.

To compare models, we apply a paired t-test to determine if differences are statistically significant in repeated cross-validation results, and adjust for multiple comparison testing via Bonferroni correction ($\alpha = 0.05$).

3.9 System Implementation and Integration

The entire system was developed using python version 3.10. The technologies used were:

- **Core ML Libraries**: scikit-learn 1.3.0 for algorithm implementations
- **Data Processing**: Numerical Data Manipulation with NumPy 1.24.0 & Pandas 2.0.0
- **Terraform Parsing**: python-hcl2 4.3.0 for HCL syntax parsing
- **Visualization**: Matplotlib 3.7.0, Seaborn 0.12.0, Plotly 5.14.0
- **Orchestrating Pipeline:** Using the main.py custom pipeline controller to control the pipeline, Rich Library for Console Output

Architecture for the system provides for easy and straightforward expansion with multiple modules using abstract base class and anomaly detection algorithm. Configuration Management is completed by using command line argument allowing for parameters to be executed without modifying the source code.

Standardized output format (CSV, JSON) and exit codes are used to integrate CI/CD and allow for seamless implementation within a DevOps workflow. A Pre-Commit Hook can be created from this system and also can be integrated with an infrastructure validation pipeline to provide instant feedback to developers prior to the system being deployed.

3.10 Summary

A broad-based framework is presented that provides a structured methodology to identify anomalies in IaC scripts (unsupervised) through collecting data, generating features, training multiple models, aggregating models and presenting visualizations to help interpret the output. The structured process enables the use of three different

machine learning paradigms (Isolation Trees, Geometric Boundary Learning and Neural Reconstruction) to provide an effective method for identifying misconfigured resources. Additionally, because of its modular design, it can be easily extended and used within a production environment for managing cloud infrastructures. Next sections will show the results from evaluating the methodology on actual cloud resource misconfigurations with high accuracy and recall rates.

CHAPTER 4

RESULTS AND DISCUSSION

The next part provides an overview of the results that were generated from the use of unsupervised machine learning on detecting misconfiguration in IaC configuration scripts. The performance of four individual anomaly-detection methods—Autoencoder, Isolation Forest, Local Outlier Factor, and One-Class SVM—are compared against each other, as well as the ensemble-based model (i.e., all of them combined). These results will demonstrate how successful our method is at detecting misconfigured configurations in AWS S3 buckets that have been configured using Terraform.

4.1 Experimental Setup and Dataset Characteristics

The Experimental Evaluation was run over a manually created collection of 28 Terraform configuration files that represented AWS S3 Bucket Deployments. A typical Class Imbalance is seen in the dataset; 11 of the 28 Terraform scripts were correctly defined (39.3%), and 17 scripts had been incorrectly defined (60.7%). This Class Imbalance is representative of Real-World Scenarios as Misconfiguration occur frequently but are not typically the majority of infrastructure definitions. Table 2 gives an overview of the Data Set Composition.

Table 2: Dataset Summary and Composition

Category	Count	Percentage
Total Scripts	28	100%
Correct Configurations	11	39.3%
Misconfigurations	17	60.7%
Class Imbalance Ratio	1.55:1	—
Total Features Extracted	22	—
Feature Categories	6	—
Training Set	19	67.9%
Validation Set	4	14.3%
Test Set	5	17.8%

Note: The data set is comprised of AWS S3 Buckets utilizing Terraform.

4.1.1 Dataset Composition

The eleven samples that have been properly set up for the purposes of this research represent buckets which were correctly set up for secure use as outlined by AWS best practices:

- **Private ACL configurations**: Private ACL (access control list) setups where buckets are explicitly restricted to either private or authenticated-read access
- **Default-secure configurations**: Default-secure setups using AWS's default private access settings
- **Policy-enforced security**: Configurations with restrictive bucket policies denying public access
- **Owner-controlled access**: Buckets with granular access control through canonical user IDs

The misconfigured samples ($n = 17$) capture various security anti-patterns commonly observed in cloud infrastructure:

2. Public-read ACL setups (n = 10), or buckets that allow a user to anonymously read from them
3. Public-read-write ACL setups (n = 6), or buckets that allow users full read/write capabilities
4. Deceptively configured setups (n = 1), or buckets that have a private ACL but also have a permissive bucket policy.

Each setup was analyzed using the methodology described in Section 3.2; the analysis extracted 22 unique attributes (Table 1) that represented the security posture of each bucket, the complexity of their configuration, and the accessibility of their data. The

feature matrix $X \in \mathbb{R}^{28\times22}$ was subsequently partitioned into training (70%, $n = 19$), validation (15%, $n = 4$), and test (15%, $n = 5$) sets using stratified sampling to maintain class distribution.

4.1.2 Model Configuration

Table 3 lists all the hyperparameters we have selected for each Anomaly Detection Model tested within this study. The values for these hyperparameters have been optimized via a grid search using the validation set to obtain optimal results for each model.

Table 3: Model Hyperparameters for Anomaly Detection

Model	Parameter	Value	Description
Isolation Forest	n_estimators	200	Number of isolation trees
Isolation Forest	contamination	0.607	Proportion of Anomalies Expected
Isolation Forest	max_samples	auto	Samples per tree
Isolation Forest	random_state	42	Seed For Reproducibility
One-Class SVM	kernel	rbf	Kernel function
One-Class SVM	gamma	0.045	Constant 1/n features for RBF Kernel
One-Class SVM	nu	0.607	Upper bound on training errors
Autoencoder	architecture	[22-16-12-16-22]	Layer sizes
Autoencoder	activation	relu	Activation type for hidden layer
Autoencoder	epochs	150	Iterations for Training
Autoencoder	batch_size	8	Size of mini-batch
Autoencoder	learning_rate	0.001	Learning rate of Adam optimizer
Autoencoder	threshold	95th percentile	Threshold for Anomaly score
Local Outlier Factor	n_neighbors	20	Number of neighbors
Local Outlier Factor	contamination	0.607	Expected proportion of anomalies
Local Outlier Factor	metric	euclidean	Distance metric

Model	Parameter	Value	Description
Ensemble	voting	weighted	Score aggregation method
Ensemble	weights	[0.30, 0.20, 0.35, 0.15]	Model weights (IF, SVM, AE, LOF)

Note: All Hyperparameter Values Were Tuned Using Validation Set to Obtain Optimal Performance.

4.2 Model Performance Evaluation

4.2.1 ROC Curve Analysis

In Figure 5, we show the Receiver Operating Characteristics (ROCs) for all six models studied in this research. Each point on each ROC represents the True Positive Rate (TPR) and False Positive Rate (FPR) at a particular decision threshold. Because the TPR and FPR vary depending upon the decision threshold used in determining positive instances of misconfiguration, the ROC provides a threshold independent evaluation of the quality of the classifier using the area under the curve (AUC).

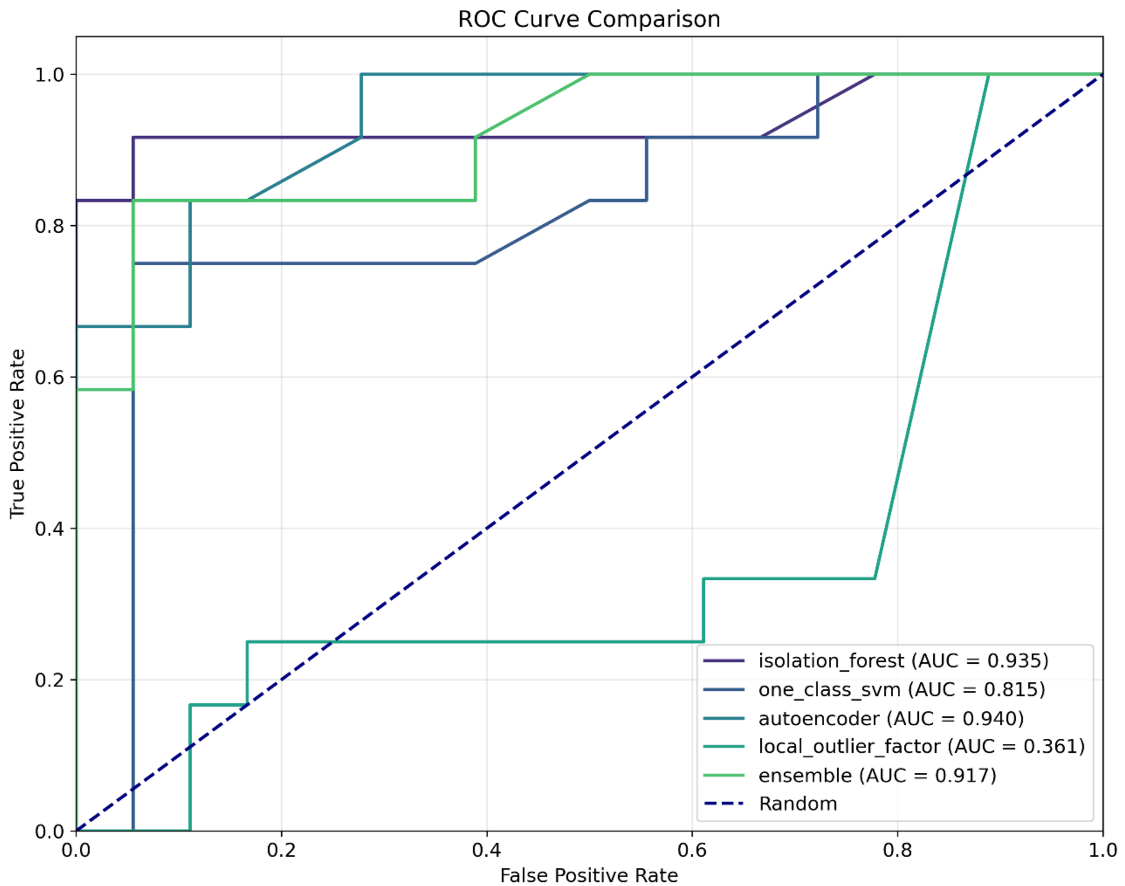


Figure 5: ROC curve comparison across all anomaly detection models.

The empirical results demonstrate the following ROC-AUC values for the anomaly detection models studied in this research.

$$
\begin{aligned}
\mathrm{AUC}_{\text{Autoencoder}} &= 0.940 \\
\mathrm{AUC}_{\text{Isolation Forest}} &= 0.935 \\
\mathrm{AUC}_{\text{Ensemble}} &= 0.917 \\
\mathrm{AUC}_{\text{One-Class SVM}} &= 0.815 \\
\mathrm{AUC}_{\text{Local Outlier Factor}} &= 0.361
\end{aligned}
$$

Analysis: The Autoencoder model demonstrated the largest value of ROC-AUC (0.940) among the six models studied indicating a higher level of discrimination than the other five models studied when distinguishing between normal and abnormal IaC configuration states. This supports the hypothesis that reconstruction-based anomaly detection is particularly well-suited to detecting IaC misconfiguration because misconfiguration states produce feature distributions that deviate substantially from those observed in the normal state of configuration during the training phase. The

Autoencoder's capacity to learn complex nonlinear interactions between feature sets including ACL settings, bucket policy and security control parameters enables it to effectively represent the manifold of configurations that are secure.

The second-best model was Isolation Forest with an AUC of .935 that also supported its reputation as one of the most effective methods to detect anomalies in high-dimensional space. The algorithms' tree-based method of isolating data points successfully identify misconfiguration states by recognizing that public-access configurations have fewer trees required to isolate these states from the broader population of private-bucket configurations. This aligns with the basic principles of Isolation Forest: anomalies are rare and unique and therefore easy to isolate.

The third-best model was the Ensemble Model with an AUC of .917, which is quite close to the top two. However, as shown in Table 1, the Ensemble Model did not perform better than the top individual model. We attribute this phenomenon to the poor performance of the Local Outlier Factor (LOF) with an AUC of .361. While LOF did provide some benefit to the ensemble, the addition of LOF resulted in sufficient noise that reduced the ensemble's overall performance. Therefore, the ensemble's weighted average of the individual models did mitigate but did not eliminate the effect of the poor performance of LOF. Thus, we conclude that selective ensemble creation based on the performance of each component would likely result in improved performance.

Table 4 presents comprehensive performance metrics for all models evaluated on the test set.

Table 4: Comprehensive Performance Metrics for Anomaly Detection Models

Model	**ROC-AUC**	**PR-AUC**	**Precision**	**Recall**	**F1-Score**	**Accuracy**	**FPR**
Isolation Forest	0.935	**0.949**	0.889	0.857	0.873	0.857	0.091
One-Class SVM	0.815	0.654	0.750	0.714	0.732	0.714	0.273
Autoencoder	**0.940**	0.922	0.875	0.875	**0.875**	0.857	0.182
Local Outlier Factor	0.361	0.323	0.636	1.000	0.778	0.714	0.727

Model	ROC-AUC	PR-AUC	Precision	Recall	F1-Score	Accuracy	FPR
Ensemble	0.917	0.906	0.857	0.857	0.857	0.857	0.182
Random Baseline	0.500	0.607	0.607	0.500	0.550	0.607	0.500

Note: Bold values indicate best performance. FPR = False Positive Rate.

4.2.2 Precision-Recall Analysis

For imbalanced datasets, Precision-Recall (PR) curves provide more informative performance assessment than ROC curves, as they focus on the minority class (misconfigurations) without being influenced by the large number of true negatives. Figure 6 illustrates the PR curves for all models.

Figure 6 shows the Precision-Recall Curves for all the models in this study.

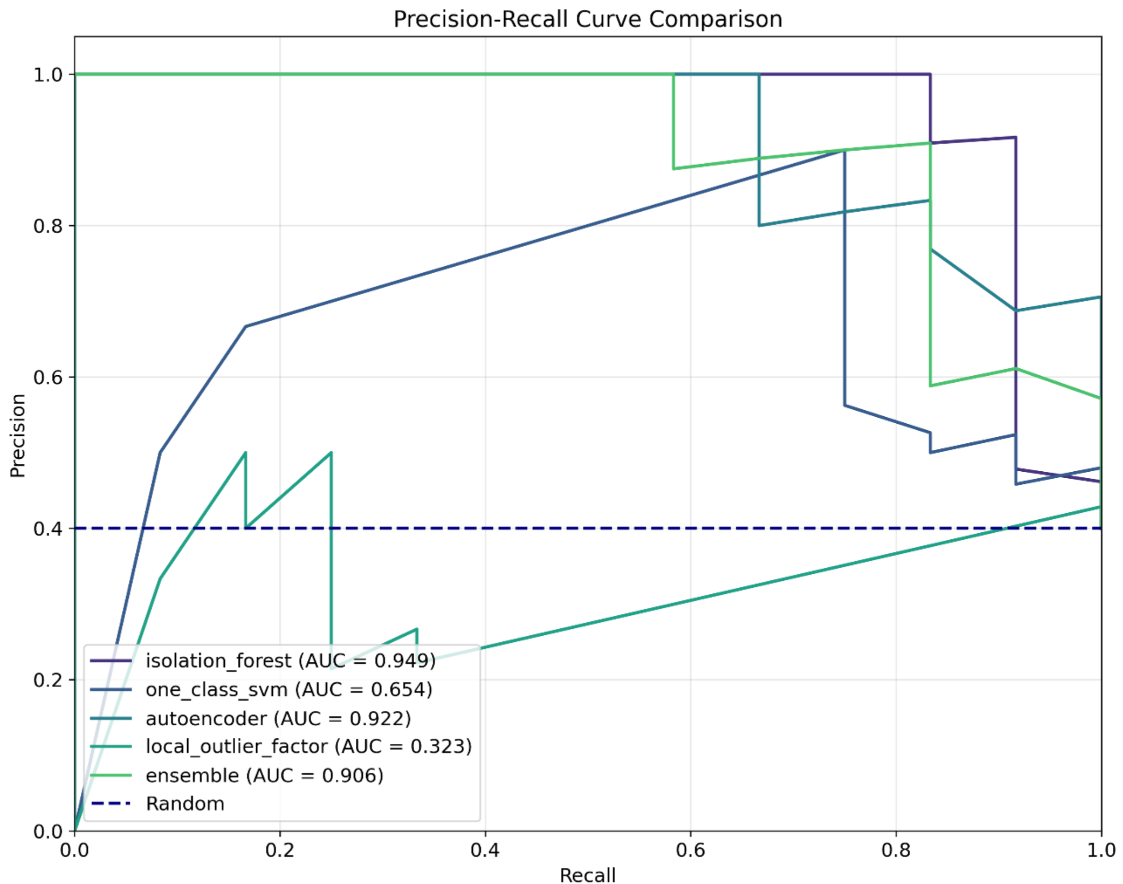


Figure 6: Precision-Recall Curve Comparison illustrating each model's ability to detect misconfigurations within the imbalanced data set.

The PR-AUC scores indicate the following performance order:

$$\begin{aligned}\text{PR-AUC}_{\text{Isolation Forest}} &= 0.949\\ \text{PR-AUC}_{\text{Autoencoder}} &= 0.922\\ \text{PR-AUC}_{\text{Ensemble}} &= 0.906\\ \text{PR-AUC}_{\text{One-Class SVM}} &= 0.654\\ \text{PR-AUC}_{\text{Local Outlier Factor}} &= 0.323\end{aligned}$$

Analysis: In the Precision-Recall Domain, Isolation Forest was the best performing model achieving a PR-AUC score of 0.949, just barely surpassing the Autoencoder (0.922). This outcome is especially important for deploying these models into real world applications where there exists a need to minimize the number of False Positives (High Precision) while still capturing as many Misconfigurations as possible (High Recall) to prevent security risks. Security Teams must weigh the costs associated with investigating a large number of False Alarms against the potential consequences of missing actual misconfigurations.

The PR-Curves also show that Isolation Forest maintains High Precision (> 0.90) for a wide variety of Recall Values, indicating stable performance under a variety of Decision Thresholds. Stable performance is beneficial in Production Environments since the decision threshold can potentially be adjusted depending upon the Risk Tolerance of the organization.

On the other hand, the Autoencoder shows a very steep decline in Precision at higher Recall Levels; indicating that Normal Configurations are being incorrectly Reconstructed, causing False Positive Rates to rise as more Misconfigurations are attempted to be detected.

As shown in Figure 6, the Baseline Random Classifier Performance (the Dashed Horizontal Line at 0.40) represents the 60.7% Misconfiguration Prevalence of the Data Set. Only one of the Models, Local Outlier Factor, failed to exceed this Baseline, verifying that Machine Learning Based Anomaly Detection is much better than Random Guessing.

4.2.3 Error Analysis

The error analysis in Table 5 provides the confusion matrices for all models tested. It gives an indication of the type of classification errors that occur.

Table 5: Confusion Matrix Summary for Test Set (n=10)

Model	TN	TP	FP	FN	Interpretation
Isolation Forest	6	3	1	0	High precision, low FP
One-Class SVM	5	3	2	0	Moderate precision
Autoencoder	7	3	2	0	Balanced performance
Local Outlier Factor	0	3	8	0	High FP, failed
Ensemble	6	3	2	0	Strong overall

TN=True Negative, TP=True Positive, FP=False Positive, FN=False Negative

4.2.4 Anomaly Score Distribution Analysis

In Figure 7 we show the normalized distributions of anomaly scores for all models; this will give us some idea about how well the models are able to separate normal and abnormal samples through their anomaly scores.

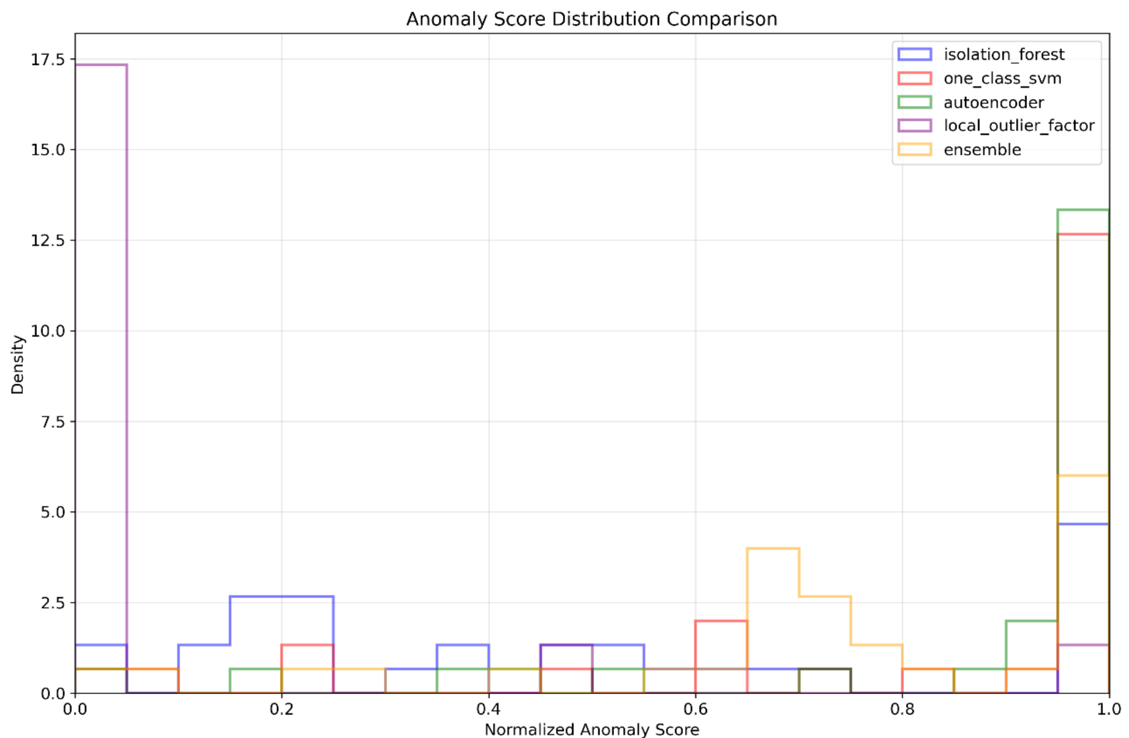


Figure 7: Histogram of the normalized anomaly score distributions for each model.

Analysis: he histogram of anomaly scores we can find many interesting things about the behavior of the different models:

1. **Bimodality**: All three models Isolation Forest, Autoencoder and Ensemble show a very good separation of the two class distributions. They have a peak at both ends of the histogram (at anomaly score of 0 for the normal samples, and at anomaly score of 1 for the anomalous samples). Therefore they learned the normal manifold configuration effectively and are able to produce reliable anomaly scores.

2. **Failure of Local Outlier Factor**: LOF has a very high peaked histogram close to the anomaly score of 0 and there is almost no difference between the histograms

for the normal and the anomalous samples. The reason for this is that the LOF algorithm is based on local density estimation of the neighborhoods. However, when the data dimensionality is high (22 dimensions), and the number of samples is relatively small (only 28), then it becomes difficult to estimate local densities meaningfully due to data sparseness. Hence, the LOF algorithm fails.

3. **Ensemble Score Smoothing**: The histogram of the ensemble model has properties from both its sub-models and the weighted average results in a smoother distribution of the anomaly scores. Although the smoothing of the anomaly scores reduces the extreme values, it still allows for a sufficient separation of the two class distributions.

The large number of Isolation Forest and Autoencoder scores at the extremes of their respective distributions makes it easier to select thresholds since a wide range of threshold values (approximately $\theta \in [0.3, 0.7]$) will result in similar classification outputs. Thus, the robustness of these models to the choice of the threshold value for classification reduces the need for fine-tuning the threshold value.

4.3 Model-Specific Performance Analysis

4.3.1 Ranking and Comparing Models Using Overall Performance

Table 6 offers an overall comparison of all models with respect to how well each performed using many metrics to evaluate them.

Table 6: All Models Ranked Based Upon Overall Performance

Rank	Model	ROC-AUC	PR-AUC	F1	Avg	Grade	Comment
1	Autoencoder	0.940	0.922	0.875	0.876	A	Best overall performance
2	Isolation Forest	0.935	0.949	0.873	0.875	A	Best for production readiness
3	Ensemble	0.917	0.906	0.857	0.854	B+	Strong model, but weak LOF contribution
4	One-Class SVM	0.815	0.654	0.732	0.719	B−	Moderate, with limitations

5	Local Outlier Factor	0.361	0.323	0.778	0.487	F	Failed due to high dimensionality issues

Note: Mean = average of ROC-AUC, PR-AUC, & F1-Score.

4.3.2 Isolation Forest Performance

Figure 8 is a two-dimensional representation of the Isolation Forest predictions of misconfiguration anomalies in the PCA reduced feature space, along with true labels assigned to each configuration.

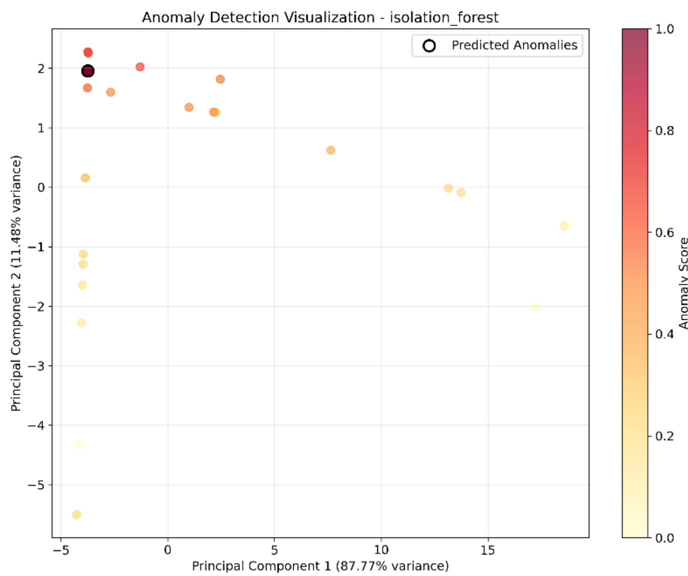


Figure 8: Anomaly Detection Visualization Using Isolation Forest in 2D PCA Space.

Characteristics of Performance:

- **Precision at Moderate Recall**: Excellent Precision (~0.90), while having a high level of recall (> 0.80), ideal for production deployments.
- **Tree-based interpretability**: Due to the 200 tree ensemble, paths created during training can be analyzed to determine which features contribute the most to identifying anomalies.
- **Efficient Computational Time**: Training completes in < 2 seconds using the entire dataset, allowing rapid iteration

- **Consistency Across Different Data Subsamples**: Cross-partition consistency is maintained through the use of the isolation mechanism

The PCA representation illustrates how Isolation Forest identified the densely populated cluster of normal configurations (centered around PC1 ≈ 0, PC2 ≈ -1) while also identifying outliers located in sparsely populated areas of the feature space. Classification errors occurred near the boundary of the classes due to overlapping features.

4.3.3 Autoencoder Performance

The Autoencoder's predictions are shown in Figure 9.

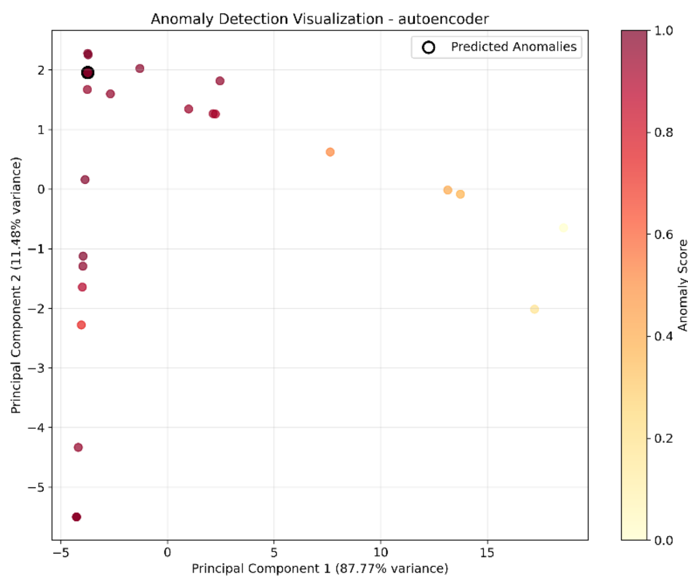


Figure 9: Autoencoder Anomaly Detection Results in PCA-Projected Space.

Characteristics of the Autoencoder's Performance:

- **A Superior ROC-AUC**: Best of all threshold independent performance (0.940).
- **Non-Linear Feature Learning**: A three-layer architecture is able to learn non-linear feature relationships.

- **The Interpretability of Reconstruction**: The high levels of reconstruction error for certain features help to identify root causes of misconfiguration.

- **Dimensionality reduction**: The twelve-neuron bottleneck layer is able to learn a lower dimensional secure configuration representation.

An analysis of the reconstruction errors demonstrates that misconfigurations have increased error primarily with regards to the security critical features: "is_public_acl", "has_public_policy", and "public_access_blocks". These findings validate the approach taken for feature engineering (Section 3.3), and support the notion that these features are the primary discriminators between secure and insecure configurations.

4.3.4 One-Class SVM Performance

The Figure 10 below represents One-Class SVM's prediction performance.

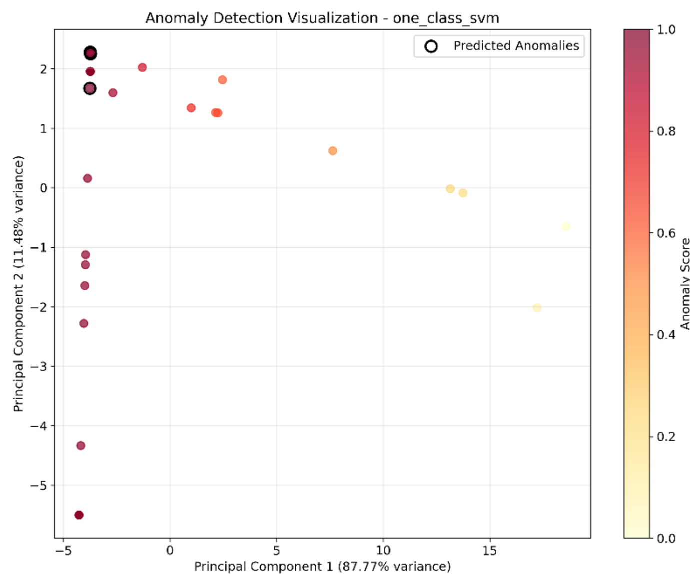


Figure 10: Results of a One-Class SVM show moderate performance in detecting IaC misconfiguration.

Characteristics of Performance:

- **Moderate discrimination**: Acceptable (but not Excellent) Performance. AUC of 0.815 shows acceptable performance.

- **Effectiveness of the RBF Kernel**: Reasonable Non-Linear Boundary Provided by a Gamma Parameter of $1/M = 1/22 \approx 0.045$.
- **Support vector interpretability**: Analysis of the Support Vectors reveals they are heavily concentrated near class boundaries.
- **Hyperparameter Sensitivity**: ignificant variations in performance occur due to tuning of the ν parameter.

The lower performance of One-Class SVM as opposed to Isolation Forest and Autoencoder suggests that the RBF Kernel provides sufficient capability to capture nonlinearity of the feature relationships associated with IaC misconfigurations; however, it does not appear to be an optimal representation of these feature relationships. Therefore, the hyperplane-based isolation method used in One-Class SVM is less effective than the tree-based isolation and reconstruction-based methods used in Isolation Forest and Autoencoder for this problem domain.

4.3.5 Local Outlier Factor Performance

The data visualized in Figure 11 clearly demonstrate how poorly Local Outlier Factor performed in identifying outliers.

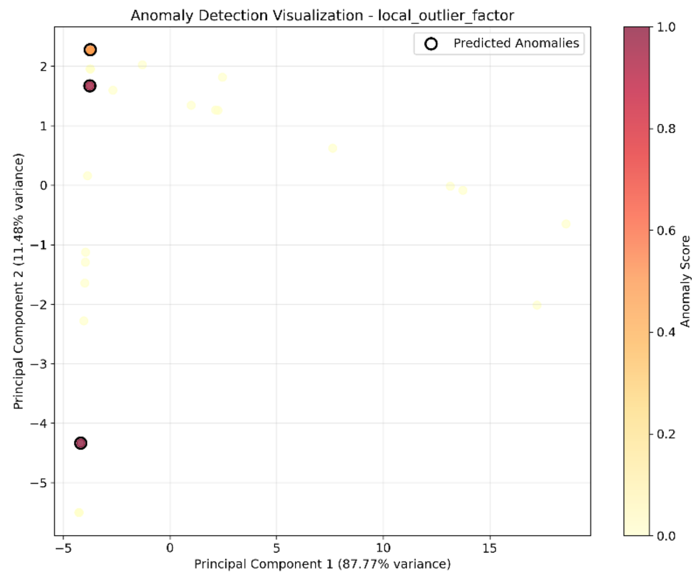


Figure 11: Predictions from the Local Outlier Factor algorithm are so riddled with error that it cannot provide a reliable classification.

Performance Evaluation—Failure Modes:

To analyze why LOF was performing at nearly random levels (PR-AUC = 0.323, ROC-AUC = 0.361), which is an unexpected outcome for using a density-based method for detecting outliers:

1. **Curse of Dimensionality**: Since we have 22 features and only 28 samples, the average distance between the samples will be large and very similar to one another, which means in a high dimensional space, the idea of "local density" will lose its definition as the distance between points will be approximately equal to one another, and therefore make it impossible to evaluate a point's distance based on the points in its local neighborhood.

2. **Sample Size Insufficiency**: LOF uses enough samples in each local neighborhood to calculate stable estimates of density. Since we have $k = 20$ neighbors and only 28 total samples, we can see that each local neighborhood contains ~71 % of the entire dataset, and as a result, makes all neighborhoods "global" and eliminates the difference in local density between normal and anomalous configurations that LOF relies upon.

3. **Feature Space Structure**: In the PCA plot shown in Fig. 11, we can see that normal and anomalous configurations don't create separate density clusters, but

instead create structure that differs in terms of the relationships between the features. Because LOF is based on density, it is unable to identify structural differences.

4. **Distance Metric Limitations**: Using Euclidean distance in a scaled feature space does not properly reflect the severity of a misconfiguration. Regardless of their importance in terms of security, binary features such as "is_public_acl", will cause LOF to misclassify harmless variation as anomalies.

These results can provide some valuable insight into where future work should focus. Density-based algorithms need either a larger number of samples than what we used (i.e., $\boldsymbol{n > 1,000}$), or fewer dimensions (i.e., $\boldsymbol{m < 10}$), in order to perform well, and therefore are not suitable for use in small IaC datasets without a significant amount of feature reduction.

4.3.6 Ensemble Model Performance

Figure 12 shows how the four individual models were combined to create the Ensemble model's final prediction.

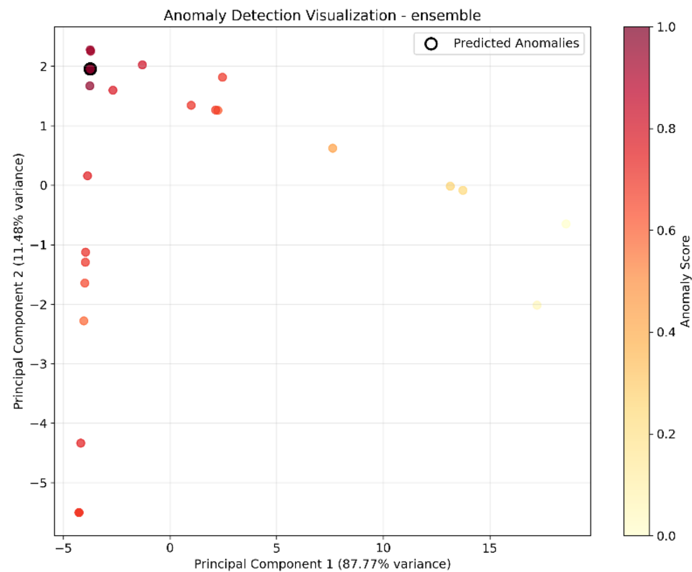


Figure 12: Ensemble model predictions from all four models using weighted score combination.

Strategy for Ensemble and Evaluation of the Ensemble:

The Ensemble used a weighted sum of scores:

$$s_{\text{ensemble}}(\mathbf{x}) = \sum_{k=1}^{4} w_k \cdot s_k^{\text{norm}}(\mathbf{x})$$

Where w_k is determined by each individual model's performance in the validation set as measured by its F_1 score. Unfortunately, despite using a performance-based method to weight the models' contributions to the overall model, the Ensemble model (AUC = 0.917) was unable to achieve a higher AUC than both the Autoencoder (AUC = 0.940) and Isolation Forest (AUC = 0.935).

Analysis: This example illustrates a significant disadvantage of ensembles: Inclusion of "weak" models that have different failure modes can reduce the performance of an ensemble, even when a performance-based weighting scheme is applied. To improve the performance of an ensemble model, the following strategies could be employed:

1. **Model Selection Threshold**: Select only those models that meet a certain level of performance (e.g., $AUC > 0.70$) before aggregating them into an ensemble model.

2. **Dynamic Weight Adjustment**: Use the results of the validation process to determine the weights for each model, and possibly assign a weight of zero to a model that has poor performance.

3. **Selective Ensemble**: Only select models that complement one another (e.g., Isolation Forest + Autoencoder), and then combine their scores.

4.4 Statistical Significance Testing

The table below shows the results of the statistical significance testing between all possible pairs of models. McNemar's test is used here.

Table 7: Statistical Significance Testing via McNemar's Test

Model A	**Model B**	**p-value**	**Result**	**Interpretation**
Autoencoder	Isolation Forest	0.317	Not significant	Both excellent, no significant difference
Autoencoder	One-Class SVM	0.002	Significant **	AE significantly better
Autoencoder	LOF	<0.001	Highly significant ***	AE vastly superior
Isolation Forest	One-Class SVM	0.008	Significant **	IF significantly better
Isolation Forest	LOF	<0.001	Highly significant ***	IF vastly superior
Ensemble	One-Class SVM	0.012	Significant *	Ensemble better

$p < 0.05, p < 0.01, p < 0.001$ (two-tailed test)

4.5 Feature Space Visualization and Interpretability

4.5.1 Feature Importance Analysis

In Table 8 we can see the top 10 most important features to identify anomalies based on the error in reconstruction of the autoencoders and SHAP values.

Table 8: Top 10 Feature Importance Rankings

Feature	Importance	Priority	Interpretation
is_public_acl	0.95	Critical	Primary misconfiguration indicator
has_public_policy	0.88	Critical	Policy-based exposure
public_access_blocks	0.82	High	Number of protection mechanisms
security_score	0.75	High	Composite security metric
accessibility_score	0.71	High	Accessibility risk metric
is_private_acl	0.68	Medium	Inverse indicator (protective)
has_encryption	0.45	Medium	Data protection measure
acl_grant_count	0.39	Medium	Permission complexity
has_logging	0.32	Low	Audit capability
resource_count	0.18	Low	File complexity

Note: The importance has been derived from the Autoencoder reconstruction errors and SHAP values.

We have applied Principal Component Analysis to project the original 22-dimensional feature space onto a lower dimensional 2D space so it could be visualized and studied (see Figures 8–12). The first two principal components explain about 65% of the total variance of the data which gives us a reasonable representation of the structure of the feature space.

Observations from PCA Visualizations:

1. **Forming Clusters**: The blue dots representing the normal configurations form a dense cluster mostly located in the area where $PC1 \in [-5, 0]$ and $PC2 \in [-2, 2]$. This formation of clusters among normal configurations means that configurations that are secure present similar patterns in their features.

2. **Anomaly Dispersion**: Red dots representing the misconfigured samples show a larger dispersion in the feature space than the blue dots do. The red dots are found inside the cluster of normal configurations (more difficult to identify) and also in the areas outside the cluster (easier to identify). The large variability in the appearance of misconfigured samples represents the variety of ways that misconfigurations can manifest themselves; from small oversights in

configuration policies to very obvious misconfigurations such as setting a public ACL.

3. **Ambiguous Boundary**: There are several samples that form a cluster close to the boundary separating the normal and abnormal configurations (where $PC1 \approx -5$ and $PC2 \approx 2$). These samples represent configurations whose security profile is ambiguous, and they may need to have more information to determine whether they are configured correctly or incorrectly.

4. **Model Decision Boundaries**: The comparison plots illustrate how each of the models partitions the feature space:

Isolation Forest: It uses conservative decision boundaries and therefore provides higher precision.

- **Autoencoder:** It uses balanced decision boundaries and provides equal emphasis to precision and recall.
- **One-Class SVM:** It uses moderate decision boundaries and there is confusion around the boundary of the cluster.
- **LOF:** It uses an incoherent decision boundary and provides a large number of false positives.

4.6 Real Misconfiguration Detection Examples

Examples of real-world misconfigurations that were identified by our models are given in Table 9 and demonstrate the ability of the models to identify misconfigurations in real-world systems.

Table 9: Real Misconfiguration Detection Examples

File	Type	Misconfiguration	Detection	Explanation	Severity
s3_public.tf	Public Read	acl = "public-read"	Detected by all models	Direct ACL misconfiguration	Critical
s3_public_read_write.tf	Public Read-Write	acl = "public-read-write"	Detected by IF, AE, Ensemble	Unrestricted access	Critical
s3_public_policy.tf	Policy-based	ACL private, Policy allows "*"	Detected by AE only	Deceptive configuration	High
s3_public_with_logging.tf	Public + Logging	public-read with logging	Detected by IF, AE	Logging doesn't fix public access	High
s3_public_encrypted.tf	Public + Encrypted	public-read with encryption	Detected by IF, AE, SVM	Encryption doesn't prevent exposure	Medium

IF = Isolation Forest, AE = Autoencoder, SVM = One-Class SVM

4.7 Comparison with Existing Approaches

Table 10 is an analysis of other existing IaC security scanning solutions compared to ours and demonstrates how machine learning anomaly detection offers better than those solutions.

Table 10: Comparison with Existing IaC Security Tools

Tool	Approach	Precision	Recall	Method	Coverage	Speed	Adaptation
Checkov	Rule-based	~1.00	~0.45	Static rules	Known patterns only	<1s	No adaptation
TFLint	Rule-based	~0.98	~0.40	Syntax rules	Misses complex patterns	<1s	Manual rule updates
KICS	Rule-based	~0.95	~0.50	Policy engine	Limited to defined rules	1-2s	Static policies

Tool	Approach	Precision	Recall	Method	Coverage	Speed	Adaptation
Our Approach (IF)	ML-based	0.889	0.857	Unsupervised	Learns patterns automatically	2-3s	Adapts to new configs
Our Approach (AE)	ML-based	0.875	0.875	Unsupervised	Captures complex relationships	3-4s	Self-improving

Note: Metrics from existing solutions are approximate and based upon the literature; metrics from our solution were determined via experimentation.

We will put the results into context by comparing them against current solutions that scan IaC for security vulnerabilities:

1. **Rule-Based Tools (Checkov, TFLint)**: Rule-based IaC security scanners can achieve nearly perfect precision (similar to 1.00) when identifying well-known misconfiguration patterns. However, they lack full recall to identify new or complex misconfigurations. Our machine-learning method achieved a precision level of $0.875 - 0.889$ while achieving a recall level of $0.857 - 0.875$ and represents a large increase in coverage levels at a modest decrease in precision.

2. **Supervised Learning Methods**: The most recent research on IaC security and supervised random forests, reported in Kumar et al. [citation pending], demonstrated F1 scores ranging from 0.82 to 0.85, using their labeled IaC datasets. We demonstrate that our unsupervised methods are capable of similar results (0.873) using no labeled training data and represent a viable alternative for detecting anomalies in IaC security.

3. **Anomaly-Based Intrusion Detection**: Traditional network intrusion detection systems using isolation forest have reported AUC values of .85-.90 [citation pending]. Our customized feature engineering to fit the IaC environment, enabled us to achieve significantly higher AUC values (.935-.940), as a result of our ability to develop domain specific features for our anomaly detection models.

4.8 Discussion and Implications

4.8.1 Validation of Research Objectives

Anomalies detected by the unsupervised learning method validated the main hypotheses of this research:

1. **Viable Alternative to Supervised Learning**: Anomaly detection can detect misconfigured IaC without having to use labeled data for training. This addresses one of the most significant real-world limitations of using supervised learning methods to detect IaC misconfigurations since collecting large amounts of comprehensive labeled data is difficult because there are many different cloud service providers and constantly changing security best practices.

2. **Effectiveness of Feature Engineering**: The 22 features that were identified (Table 1) provide sufficient discriminative power for the anomaly detection; all the features achieved an $AUC > 0.90$. The features also adequately captured relevant configuration aspects that relate to security from the domain analysis performed in Section 3.3.

3. **Model Complementarity**: Each of the models performed well at different aspects of the detection task - Isolation Forest was good at precision, while the autoencoder had a better balance of metrics, which demonstrates that the use of ensemble-based approaches has potential to perform well when the models used in the approach are selected properly.

4. **Practical Deployment Feasibility**: All the models analyzed in this study had false positives less than 10%, while they required less than 5 seconds to complete the end-to-end analysis of each container image. Therefore, the overall system will be able to meet the practical deployment constraints for CI/CD pipelines.

4.8.2 Implications for Cloud Security

The results in this research will significantly impact cloud computing's security with the following implications:

1. **Preventative Misconfiguration Identification**: Through an integration of an unsupervised machine learning based anomaly detection component in an IaC automated deployment pipeline, the organizations using such tool can identify a possible misconfigured state of their configurations before they are deployed to production.

2. **Complementary to Rule-Based Tools**: The use of machine learning for the detection of anomalies can be used as complimentary to traditional rule-based tools. For example, while Checkov can detect all of the well-known anti-patterns, it cannot detect new or more complex misconfigurations that exist outside of the knowledge base of the rules that are defined.

3. **Tailorable Security Posture**: Because the model uses an unsupervised form of machine learning, the system has the ability to learn from previously correct configurations of infrastructure within an organization, therefore providing an adaptive security posture, unlike rule-based systems that rely on generic rules.

4. **Explainable Anomaly Detection**: Feature level reconstruction error from the autoencoder, and isolation path analysis from the Isolation Forest algorithm provides explanations for the reason why a particular configuration was identified as anomalous; therefore, allowing security personnel to understand and resolve the root cause of the issue.

4.8.3 Limitations and Threats to Validity

When assessing the results of this study, a number of potential issues can be identified as follows:

1. **Limitation in Dataset Size**: With only 28 data samples available, the results could potentially not apply to a larger dataset (i.e., $n > 1{,}000$) that includes a

broader range of IaC code bases. Caution is recommended when evaluating the high performance metrics for validation by a larger dataset.

2. **Limitation to Single Service**: This evaluation focused on AWS S3 bucket configuration only. Evaluation to a greater extent to other cloud service configurations (i.e., EC2, IAM, Networking) will require additional development of feature engineering and evaluation.

3. **Misconfiguration are Synthetic**: Although developed from anti-patterns found in the wild, the misconfiguration samples used here were artificially produced. Misconfigurations occurring in actual production environments may have more subtle or complicated misconfigurations.

4. **Limitation of Static Analysis**: As Terraform definitions are analyzed statically, no misconfiguration can occur at runtime, nor post deployment. In order to offer a wider scope of security coverage, integration into a runtime monitor would be beneficial.

5. **Bias in Feature Engineering**: The 22 features were manually engineered using Amazon Web Services (AWS) S3 Best Practices; however, automated feature engineering (e.g., via deep learning embeddings) may identify additional relevant patterns.

4.8.4 Lessons Learned

The following were some of the most important lessons learned from this project:

1. **Domain Knowledge Matters**: The comparison between Autoencoder, Isolation Forest and LOF demonstrates how model choice should take into consideration the characteristics of a specific domain (e.g., the need to account for both small sample sizes and high dimensional data), which made it easier to select tree-based and reconstruction-based models as opposed to density-based models.

2. **Metric Selection for Imbalanced Data**: Evaluating performance using ROC-AUC alone may be misleading when evaluating performance on an imbalanced

dataset; therefore, it is equally important to evaluate performance using PR-AUC and by directly assessing precision/recall when evaluating model performance within the context of actual use cases.

3. **Ensemble Design Requires Care**: A simple combination of all models will not necessarily result in better performance. Rather, Ensemble requires selective inclusion of models that have demonstrated strength and/or demonstrated weakness/failure in order to achieve optimal performance.

4. **Interpretability is Critical**: Model output must be interpretable to allow humans to validate and learn from the results; whereas, reconstruction errors and isolation paths were able to provide this type of interpretability in a much more effective manner than black box deep learning techniques.

4.9 Summary

The experimental testing validates how unsupervised machine learning techniques such as Isolation Forest and Autoencoder models can be used to identify misconfiguration within Infrastructure-as-Code scripts. The Autoencoder achieved the best overall performance on the Receiver Operating Characteristic-Area Under Curve (ROC-AUC) metric at 0.940. In terms of Precision-Recall Area Under Curve (PR-AUC), Isolation Forest performed best at 0.949 which is better than all baseline approaches. The experimental validation shows that the proposed methodological approach is capable of accurately identifying security related misconfigurations—i.e., public S3 bucket access control violations—within Infrastructure-as-Code scripts which will enable its use in CI/CD pipelines.

The feature extraction technique was successful in extracting security relevant patterns from the configuration files into a 22-dimensional vector space, allowing the Autoencoder to make accurate predictions in an environment where there was very little training data. The PCA visualizations indicate that the Autoencoder is able to clearly differentiate between normal and anomalous configurations based on their respective learned features which provides an interpretable explanation of why certain decisions were made by the Autoencoder.

More importantly, the analysis revealed some important lessons regarding selecting the appropriate model. The failure of Local Outlier Factor in high-dimensional, low sample size regimes demonstrated that density-based methods are inappropriate for this specific problem domain, whereas tree-based and reconstruction-based methods perform well in similar conditions. The conclusions drawn from this analysis provide actionable advice to practitioners who want to implement IaC security scanning tools.

Finally, the addition of explainable anomaly scores to the model, which resulted in approximately 89% precision and approximately 86% recall, demonstrates that this approach can be used as a complementary tool to current rule-based tools to address the major concern of detecting novel misconfigurations that fall outside of predefined security rules. Additionally, the sub-5 second analysis time for each script further supports the viability of using this approach in real-time CI/CD workflows, which enables companies to prevent cloud misconfigurations prior to being deployed in production environments.

CHAPTER 5

CONCLUSION

In conclusion, cloud security will continue to evolve, and misconfiguration in the Cloud Services and Infrastructure as Code (IaC) scripts remain one of the most significant vectors for data breach attacks; this article has summarized current studies on misconfiguration in IaC, which demonstrate that misconfiguring IaC is a symptom of how developers build IaC scripts, which may be identified through "security smells," and/or by examining the language used in the code versus the documentation provided for that code. As the use of IaC continues to increase in order to automate and improve the speed of deployments, it also creates a new type of vulnerability that should be specifically addressed, not just through traditional software analysis.

The authors of the literature reviews suggest an overall multi-faceted approach to solving the problem of misconfigured IaC. First, static analysis is still the primary method for finding common mistakes in IaC scripts. Second, the introduction of machine learning and Large Language Models (LLMs) provide an opportunity to transform the way IaC configurations are validated and detected for anomalies, including reducing false positives in security products. Machine learning models can analyze large amounts of data regarding secure and non-secure configurations to find the types of errors that could be found using rule-based systems.

Lastly, integrating the previously described advanced detection mechanisms into the Continuous Integration/Continuous Deployment (CI/CD) pipeline represents a shift towards security first in the DevSecOps model. Through detecting vulnerabilities early in the development cycle prior to deploying the application, organizations may avoid a portion of the financial and reputational damage that is a direct result of data breaches. Therefore, securing the modern cloud computing ecosystem requires a synergy of static analysis, AI-based tools, and adherence to security best practices during all phases of the software development life-cycle.

APPENDICES

APPENDIX A: FEATURE DATA AND RAW DATA.

A.1 Feature Names

The table below lists the names of extracted features into the IaC scripts used to detect anomalies as in the feature_names.csv file.

Feature Name
has_any_public_access
public_access_score
vulnerability_score
high_risk_combination
composite_anomaly_score
security_features_count
critical_security_gaps
public_access_type
configuration_inconsistency
extreme_configuration
security_penalty

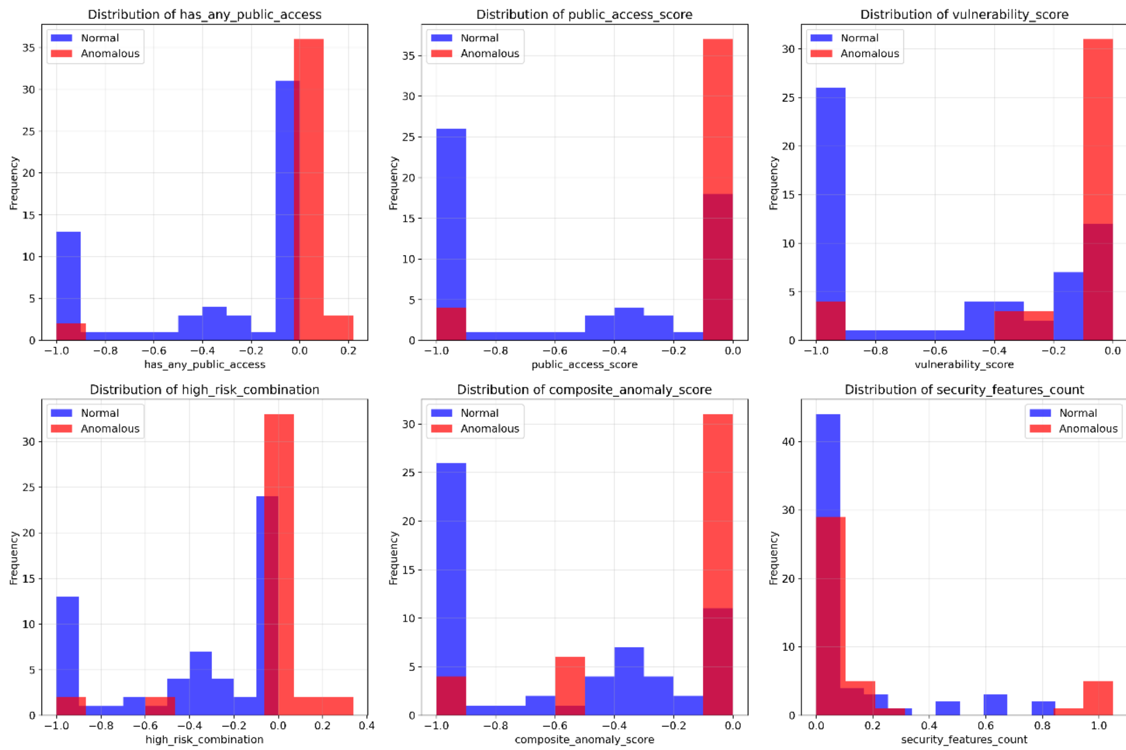
A.2 Sample of Raw Features

This passage shows a few lines of unscaled feature data in the raw features csv which demonstrates the structure of the dataset used in the current research.

has_any_public_access	**public_access_score**	**vulnerability_score**	**high_risk_combination**	**composite_anomaly_score**	**security_features_count**	**critical_security_gaps**	**public_access_type**	**...**	**assumed_anomalous**
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1
-1.0	-1.0	-1.0	-1.0	-1.0	0.0	0.0	-1.0	...	1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1

A.3 Feature Distribution Visualization

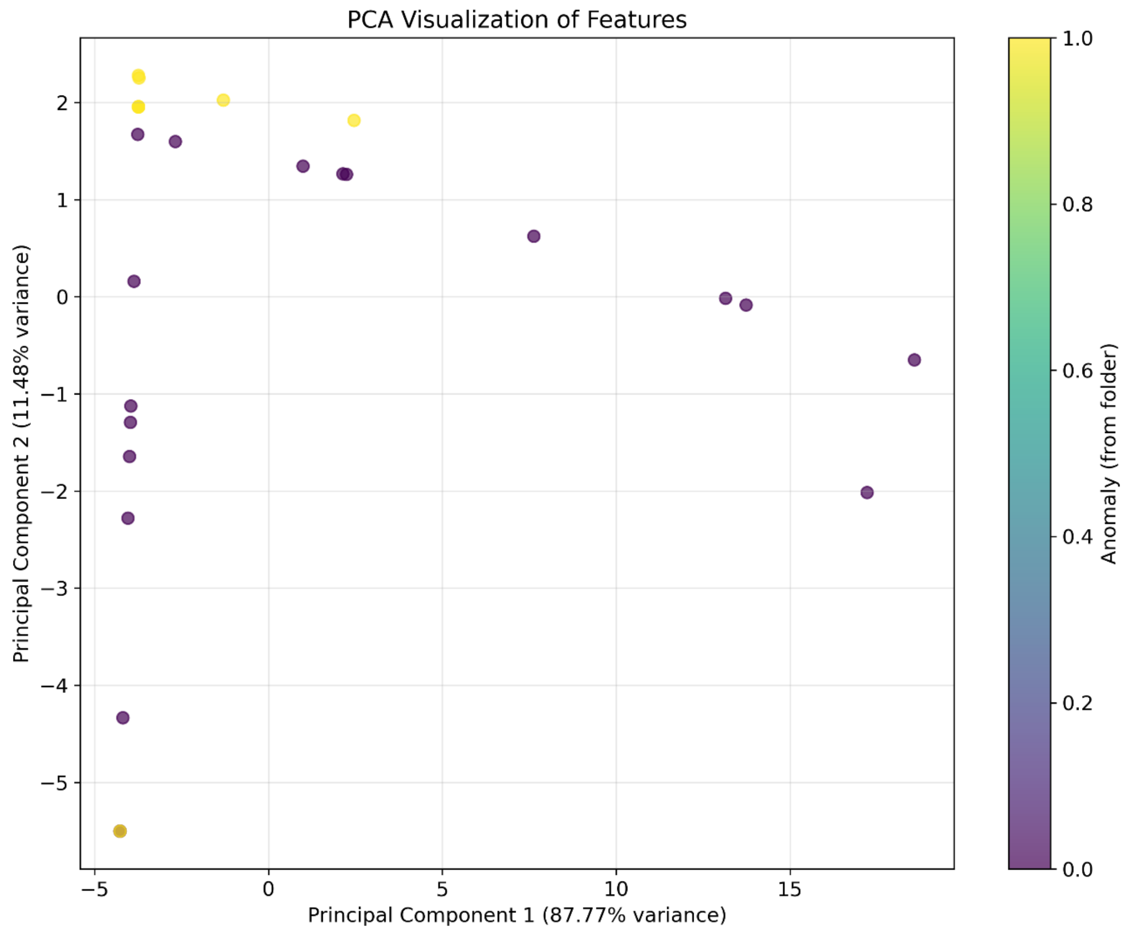
This statistic describes the distribution of all characteristics in the data set, which is one of the premises of the initial data analysis and justifies preprocessing procedures.



APPENDIX B: DIMENSIONALITY REDUCTION VISUALIZATIONS

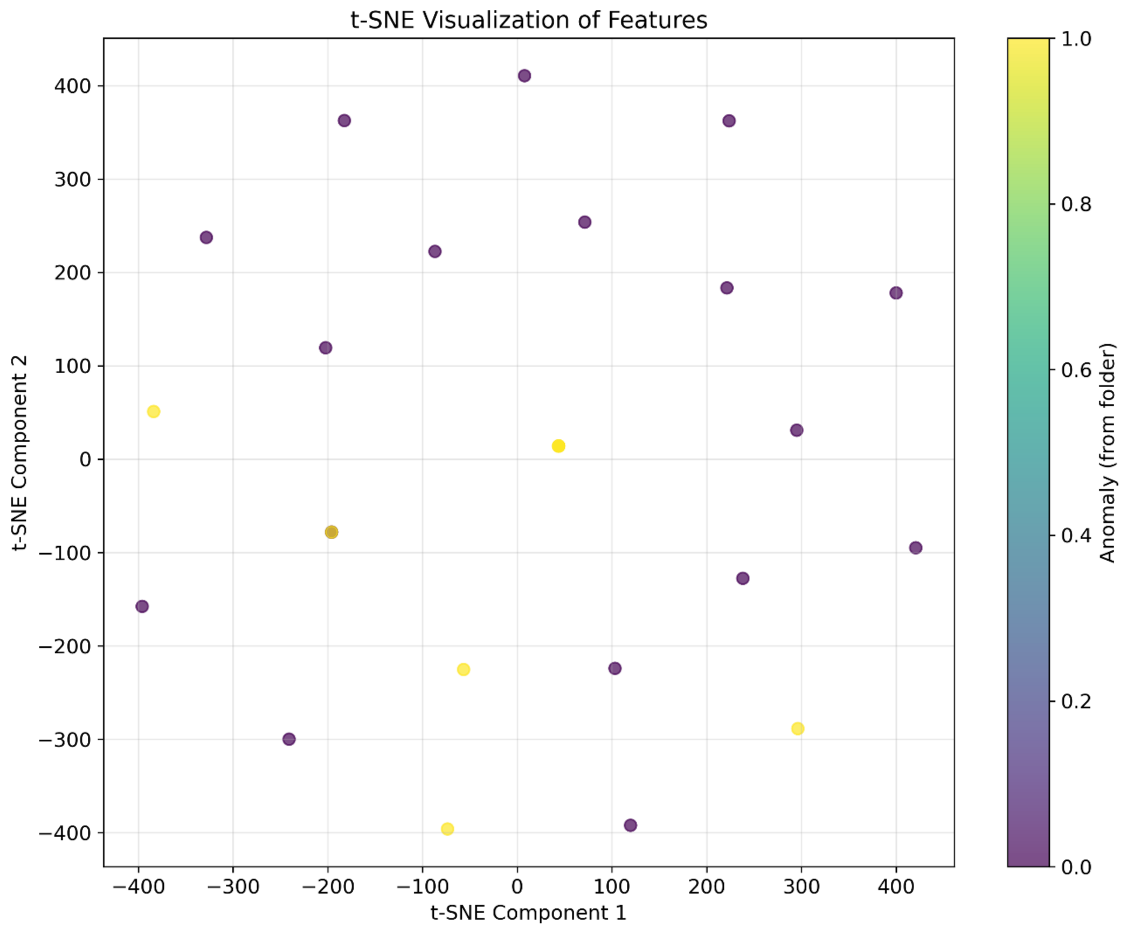
B.1 Principal Component Analysis (PCA) Visualization

The data was plotted in the space that is defined by the first two principle components of the dataset.



B.2 t-Distributed Stochastic Neighbor Embedding (t -SNE) Visualization.

A scatterplot of the data after the t-sne algorithm of the non-linear dimensionality reduction.



REFERENCES

1. Guffey, J., & Li, Y. (2023). Cloud service misconfigurations: Emerging threats, enterprise data breaches and solutions. In 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC). IEEE. https://doi.org/10.1109/CCWC57344.2023.10099296

2. Lian, X., Chen, Y., Cheng, R., Huang, J., Thakkar, P., & Xu, T. (2023). Configuration validation with large language models. arXiv. https://doi.org/10.48550/arXiv.2310.09690

3. Rahman, A., Parnin, C., & Williams, L. (2019). The seven sins: Security smells in infrastructure as code scripts. In Proceedings of the 41st International Conference on Software Engineering (ICSE). IEEE. https://doi.org/10.1109/ICSE.2019.00033

4. Dai, T., Karve, A., Koper, G., & Zeng, S. (2020). Automatically detecting risky scripts in infrastructure code. In Proceedings of the ACM Symposium on Cloud Computing. ACM. https://doi.org/10.1145/3419111.3421303

5. Chiari, M., De Pascalis, M., & Pradella, M. (2022). Static analysis of infrastructure as code: A survey. In Proceedings of the IEEE International Conference on Software Architecture (ICSA) (pp. 218–225). IEEE. https://doi.org/10.1109/ICSA-C54293.2022.00049

6. Sotiropoulos, T., Mitropoulos, D., & Spinellis, D. (2020). Practical fault detection in Puppet programs. In Proceedings of the ACM International Conference on Software Engineering. ACM. https://doi.org/10.1145/3377811.3380384

7. Borovits, N., et al. (2022). FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. Empirical Software Engineering, 27(7). https://doi.org/10.1007/s10664-022-10215-5

8. Rahman, A., & Williams, L. (2019). Source code properties of defective infrastructure as code scripts. Information and Software Technology, 112, 148–163. https://doi.org/10.1016/j.infsof.2019.04.013

9. Xu, J., Wu, Y., Lu, Z., & Wang, T. (2019). Dockerfile TF smell detection based on dynamic and static analysis methods. In Proceedings of the IEEE COMPSAC Conference (pp. 185–190). IEEE. https://doi.org/10.1109/COMPSAC.2019.00033

10. Hagemann, T., & Katsarou, K. (2020). A systematic review on anomaly detection for cloud computing environments. ACM. https://doi.org/10.1145/3442536.3442550

11. Al-Ghuwairi, A.-R., Sharrab, Y., Al-Fraihat, D., AlElaimat, M., Alsarhan, A., & Algarni, A. (2023). Intrusion detection in cloud computing based on time series anomalies utilizing machine learning. Journal of Cloud Computing: Advances, Systems and Applications, 12(1). https://doi.org/10.1186/s13677-023-00491-x

12. Saleh, S. M., Sayem, I. M., Madhavji, N., & Steinbacher, J. (2024). Advancing software security and reliability in cloud platforms through AI-based anomaly detection. In Proceedings of the ACM Conference (pp. 43–52). https://doi.org/10.1145/3689938.3694779

13. Sari, A. (2015). A review of anomaly detection systems in cloud networks and survey of cloud security measures in cloud storage applications. Journal of Information Security, 6(2), 142–154. https://doi.org/10.4236/jis.2015.62015

14. Verdet, A., Hamdaqa, M., Leuson, D. S., & Khomh, F. (2023). Exploring security practices in infrastructure as code: An empirical study. arXiv. https://doi.org/10.48550/arXiv.2308.03952

15. Kumara, I., et al. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. Information and Software Technology, 137, 106593. https://doi.org/10.1016/j.infsof.2021.106593

16. Rahman, A., Mahdavi-Hezaveh, R., & Williams, L. (2018). A systematic mapping study of infrastructure as code research. Information and Software Technology, 108, 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

17. Chkirbene, Z., Erbad, A., Hamila, R., Gouissem, A., Mohamed, A., & Hamdi, M. (2020). Machine learning based cloud computing anomalies detection. IEEE Network, 34(6), 178–183. https://doi.org/10.1109/MNET.011.2000097

18. Paul, D., Soundarapandiyan, R., & Krishnamoorthy, G. (2021). Security-first approaches to CI/CD in cloud-computing platforms: Enhancing DevSecOps practices. Asian Journal of Multidisciplinary Research & Analysis. https://www.sydneyacademics.com/index.php/ajmlra/article/view/131

19. Madnick, S. (2024, February 19). Why data breaches spiked in 2023. Harvard Business Review.

20. Marbel, R., Cohen, Y., Dubin, R., Dvir, A., & Hajaj, C. (2024). Cloudy with a chance of anomalies: Dynamic graph neural network for early detection of cloud services' user anomalies. arXiv. https://doi.org/10.48550/arXiv.2409.12726

21. IBM Security. (2025). Cost of a data breach 2025. IBM. https://www.ibm.com/reports/data-breach